Cours de C/C++

Christian Casteyde

Cours de C/C++

par Christian Casteyde

Historique des versions

Version 2.0.2 01/01/2007 Revu par : CC
Conversion du document en XML pour permettre l'utilisation des feuilles de style XML et des outils de formatage XML-FO. Corr
Version 2.0.1 26/02/2006 Revu par : CC
Corrections orthographiques. Correction sur la gestion des exceptions dans les constructeurs par les constructeurs try. Précisions su
Version 2.0.0 03/07/2005 Revu par : CC
Ajout d'une introduction sur les langages de programmation et refonte de la première partie. Ajout des définitions des termes lors d
Version 1.40.6 16/05/2004 Revu par : CC
Correction de l'exemple d'allocation dynamique de mémoire en C. Correction de l'exemple de fonction à nombre variable de para
Version 1.40.5 14/06/2003 Revu par : CC
Correction de l'allocation dynamique de tableaux à plus d'une dimension.
Version 1.40.4 21/09/2002 Revu par : CC
Correction de l'exemple de recherche sur les chaînes de caractères. Ajout des initialiseurs C99. Précisions sur la portabilité des typ
Version 1.40.3 12/05/2002 Revu par : CC
Nombreuses corrections orthographiques. Quelques corrections et précisions. Clarification de quelques exemples.
Version 1.40.2 26/01/2001 Revu par : CC
Corrections orthographiques. Ajout d'un lien sur les spécifications Single Unix de l'Open Group.
Version 1.40.1 09/09/2001 Revu par : CC
Corrections orthographiques. Précisions sur les optimisations des opérateurs d'incrémentation et de décrémentation postfixés et pré
Version 1.40.0 30/07/2001 Revu par : CC
Version finale. Réorganisation partielle de la première partie. Scission du chapitre contenant les structures de contrôle et les définiti
Version 1.39.99 24/06/2001 Revu par : CC
Description des locales standards. Précision sur l'initialisation des variables lors de leurs déclarations. Précision sur les droits d'acc
Version 1.39.4 27/05/2001 Revu par : CC
Description des flux d'entrée / sortie de la bibliothèque standard. Modification de la présentation sommaire des flux dans le chapitr
Version 1.39.3 03/05/2001 Revu par : CC
Description des algorithmes de la bibliothèque standard.
Version 1.39.2 22/04/2001 Revu par : CC
Description des conteneurs de la bibliothèque standard. Ajout d'une traduction de la licence FDL. Suppression des symboles &colo
Version 1.39.1 05/03/2001 Revu par : CC
Description des types de données complémentaires de la bibliothèque standard C++. Correction du comportement du bloc catch de
Version 1.39.0 04/02/2001 Revu par : CC
Mise en conformité des en-têtes C++ des exemples avec la norme. Correction des exemples utilisant des noms réservés par la biblio
Version 1.38.1 14/10/2000 Revu par : CC
Précisions sur les classes de base virtuelles. Corrections orthographiques.
Version 1.38.0 01/10/2000 Revu par : CC
Corrections typographiques. Précisions sur les opérateurs & et *.
Version 1.37 23/08/2000 Revu par : CC
Passage au format de fichier SGML. Ajout des liens hypertextes. Corrections mineures.
Version 1.36 27/07/2000 Revu par : CC
Complément sur les parenthèses dans les définitions de macros. Corrections sur la numérotation des paragraphes.
Version 1.35 10/07/2000 Revu par : CC
Corrections sur les déclarations using.
Version 1.34 09/07/2000 Revu par : CC
Passage en licence FDL. Ajout de la table des matières.
Version 1.33 22/60/2000 Revu par : CC
Correction d'une erreur dans le paragraphe sur les paramètres template template. Corrections orthographiques diverses.
Version 1.32 17/06/2000/ Revu par : CC
Correction d'une erreur dans le programme d'exemple du premier chapitre. Correction d'une erreur dans un exemple sur la dérivati
Version 1.31 12/02/2000 Revu par : CC
Corrections mineurs. Ajout du paragraphe sur la spécialisation d'une fonction membre d'une classe template.
Version 1.30 05/12/1999 Revu par : CC
Ajout de la licence. Modifications mineures du formatage.
Version <1.30 <1998 Revu par : CC

Version initiale.

Table des matières

Avant-propos ..... **xv**

I. Le langage C++ ..... **xvii**

1. Première approche du C/C++ 1
   - 1.1. Les ordinateurs, les langages et le C++ 1
     - 1.1.1. Les ordinateurs et la programmation 1
     - 1.1.2. Les langages de programmation 1
     - 1.1.3. Le langage C/C++ 3
     - 1.1.4. Les outils de programmation 3
   - 1.2. Notre premier programme 4
     - 1.2.1. Hello World! 4
     - 1.2.2. Analyse du programme 5
     - 1.2.3. Généralisation 6
   - 1.3. Les commentaires en C/C++ 8
   - 1.4. Les variables 9
     - 1.4.1. Définition des variables 9
     - 1.4.2. Les types de base du C/C++ 10
     - 1.4.3. Notation des valeurs 12
       - 1.4.3.1. Notation des valeurs booléennes 12
       - 1.4.3.2. Notation des valeurs entières 13
       - 1.4.3.3. Notation des valeurs en virgule flottantes 13
       - 1.4.3.4. Notation des caractères 14
       - 1.4.3.5. Notation des chaînes de caractères 15
   - 1.5. Les instructions 15
     - 1.5.1. Les instructions simples 16
     - 1.5.2. Les instructions composée 19
     - 1.5.3. Les structures de contrôle 19
   - 1.6. Les fonctions et les procédures 19
     - 1.6.1. Définition des fonctions et des procédures 20
     - 1.6.2. Appel des fonctions et des procédures 21
     - 1.6.3. Notion de déclaration 22
     - 1.6.4. Surcharge des fonctions 23
     - 1.6.5. Fonctions inline 24
     - 1.6.6. Fonctions statiques 25
     - 1.6.7. Fonctions prenant un nombre variable de paramètres 25
   - 1.7. Les entrées / sorties en C 27
     - 1.7.1. Généralités sur les flux d'entrée / sortie 27
     - 1.7.2. Les fonctions d'entrée / sortie de la bibliothèque C 28
     - 1.7.3. La fonction `printf` 29
     - 1.7.4. La fonction scanf 31
2. Les structures de contrôle 35
   - 2.1. Les tests 35
     - 2.1.1. La structure conditionnelle if 35
     - 2.1.2. Le branchement conditionnel 36
   - 2.2. Les boucles 37
     - 2.2.1. La boucle for 37
     - 2.2.2. Le while 38
     - 2.2.3. Le do 39
   - 2.3. Les instructions de rupture de séquence et de saut 39
     - 2.3.1. Les instructions de rupture de séquence 39

2.3.2. Le saut 40
3. Types avancés et classes de stockage 43
3.1. Types de données portables 43
3.2. Structures de données et types complexes 44
3.2.1. Les tableaux 45
3.2.2. Les chaînes de caractères 46
3.2.3. Les structures 47
3.2.4. Les unions 49
3.2.5. Les champs de bits 50
3.2.6. Initialisation des structures et des tableaux 51
3.3. Les énumérations 52
3.4. Les alias de types 53
3.4.1. Définition d'un alias de type 53
3.4.2. Utilisation d'un alias de type 54
3.5. Transtypages et promotions 55
3.6. Les classes de stockage 57
4. Les pointeurs et références 61
4.1. Notion d'adresse 61
4.2. Notion de pointeur 61
4.3. Déréférencement, indirection 62
4.4. Notion de référence 64
4.5. Lien entre les pointeurs et les références 64
4.6. Passage de paramètres par variable ou par valeur 65
4.6.1. Passage par valeur 65
4.6.2. Passage par variable 66
4.6.3. Avantages et inconvénients des deux méthodes 66
4.6.4. Comment passer les paramètres par variable en C ? 67
4.6.5. Passage de paramètres par référence 67
4.7. Références et pointeurs constants et volatiles 69
4.8. Arithmétique des pointeurs 72
4.9. Utilisation des pointeurs avec les tableaux 73
4.10. Les chaînes de caractères : pointeurs et tableaux à la fois ! 74
4.11. Allocation dynamique de mémoire 75
4.11.1. Allocation dynamique de mémoire en C 75
4.11.2. Allocation dynamique en C++ 80
4.12. Pointeurs et références de fonctions 82
4.12.1. Pointeurs de fonctions 82
4.12.2. Références de fonctions 84
4.13. Paramètres de la fonction main - ligne de commande 85
4.14. DANGER 86
5. Le préprocesseur C 89
5.1. Définition 89
5.2. Les directives du préprocesseur 89
5.2.1. Inclusion de fichier 89
5.2.2. Constantes de compilation et remplacement de texte 90
5.2.3. Compilation conditionnelle 91
5.2.4. Autres directives 92
5.3. Les macros 92
5.4. Manipulation de chaînes de caractères dans les macros 95
5.5. Les trigraphes 96
6. Modularité des programmes et génération des binaires 97
6.1. Pourquoi faire une programmation modulaire ? 97

6.2. Les différentes phases du processus de génération des exécutables 98
6.3. Compilation séparée en C/C++ 100
6.4. Syntaxe des outils de compilation 101
6.4.1. Syntaxe des compilateurs 101
6.4.2. Syntaxe de make 102
6.5. Problèmes syntaxiques relatifs à la compilation séparée 103
6.5.1. Déclaration des types 103
6.5.2. Déclaration des variables 103
6.5.3. Déclaration des fonctions 103
6.5.4. Directives d'édition de liens 104
7. C++ : la couche objet 107
7.1. Généralités 107
7.2. Extension de la notion de type du C 108
7.3. Déclaration de classes en C++ 108
7.4. Encapsulation des données 112
7.5. Héritage 114
7.6. Classes virtuelles 117
7.7. Fonctions et classes amies 118
7.7.1. Fonctions amies 119
7.7.2. Classes amies 119
7.8. Constructeurs et destructeurs 120
7.8.1. Définition des constructeurs et des destructeurs 121
7.8.2. Constructeurs de copie 125
7.8.3. Utilisation des constructeurs dans les transtypages 126
7.9. Pointeur this 127
7.10. Données et fonctions membres statiques 128
7.10.1. Données membres statiques 129
7.10.2. Fonctions membres statiques 130
7.11. Surcharge des opérateurs 131
7.11.1. Surcharge des opérateurs internes 132
7.11.2. Surcharge des opérateurs externes 134
7.11.3. Opérateurs d'affectation 137
7.11.4. Opérateurs de transtypage 138
7.11.5. Opérateurs de comparaison 139
7.11.6. Opérateurs d'incrémentation et de décrémentation 139
7.11.7. Opérateur fonctionnel 140
7.11.8. Opérateurs d'indirection et de déréférencement 142
7.11.9. Opérateurs d'allocation dynamique de mémoire 143
7.12. Des entrées - sorties simplifiées 150
7.13. Méthodes virtuelles 152
7.14. Dérivation 154
7.15. Méthodes virtuelles pures - Classes abstraites 156
7.16. Pointeurs sur les membres d'une classe 161
8. Les exceptions en C++ 165
8.1. Techniques de gestion des erreurs 165
8.2. Lancement et récupération d'une exception 169
8.3. Hiérarchie des exceptions 171
8.4. Traitement des exceptions non captées 173
8.5. Liste des exceptions autorisées pour une fonction 174
8.6. Gestion des objets exception 176
8.7. Exceptions dans les constructeurs et les destructeurs 177
9. Identification dynamique des types 181

9.1. Identification dynamique des types 181
9.1.1. L'opérateur typeid 181
9.1.2. La classe type_info 183
9.2. Transtypages C++ 183
9.2.1. Transtypage dynamique 184
9.2.2. Transtypage statique 186
9.2.3. Transtypage de constance et de volatilité 187
9.2.4. Réinterprétation des données 187
10. Les espaces de nommage 189
10.1. Définition des espaces de nommage 189
10.1.1. Espaces de nommage nommés 189
10.1.2. Espaces de nommage anonymes 191
10.1.3. Alias d'espaces de nommage 192
10.2. Déclaration using 192
10.2.1. Syntaxe des déclarations using 192
10.2.2. Utilisation des déclarations using dans les classes 194
10.3. Directive using 195
11. Les template 199
11.1. Généralités 199
11.2. Déclaration des paramètres template 199
11.2.1. Déclaration des types template 199
11.2.2. Déclaration des constantes template 200
11.3. Fonctions et classes template 201
11.3.1. Fonctions template 201
11.3.2. Les classes template 202
11.3.3. Fonctions membres template 205
11.4. Instanciation des template 208
11.4.1. Instanciation implicite 208
11.4.2. Instanciation explicite 209
11.4.3. Problèmes soulevés par l'instanciation des template 210
11.5. Spécialisation des template 211
11.5.1. Spécialisation totale 211
11.5.2. Spécialisation partielle 212
11.5.3. Spécialisation d'une méthode d'une classe template 214
11.6. Mot-clé typename 215
11.7. Fonctions exportées 216
12. Conventions de codage et techniques de base 217
12.1. Conventions de codage 217
12.1.1. Généralités 217
12.1.2. Lisibilité et cohérence du code 218
12.1.2.1. Règles de nommage des identificateurs 218
12.1.2.2. Règles de cohérence et de portabilité 221
12.1.2.3. Règles de formatage et d'indentation 224
12.1.3. Réduction des risques 225
12.1.3.1. Règles de simplicité 225
12.1.3.2. Règles de réduction des effets de bords 226
12.1.3.3. Règles de prévention 228
12.1.4. Optimisations 233
12.1.4.1. Règles d'optimisation générales 233
12.1.4.2. Autres règles d'optimisation 234
12.2. Méthodes et techniques classiques 236
12.2.1. Méthodologie objet 236

12.2.1.1. Définir le besoin et le périmètre des fonctionnalités 236
12.2.1.2. Identifier les entités et leur nombre 237
12.2.1.3. Analyser les interactions et la dynamique du système 238
12.2.1.4. Définir les interfaces 238
12.2.1.5. Réalisation et codage 239
12.2.1.6. Les tests 239
12.2.1.7. Les design patterns 240
12.2.2. Programmation objet en C 241
12.2.2.1. Principe de base 241
12.2.2.2. Définition d'interfaces 241
12.2.2.3. Compatibilité binaire 244
12.2.3. ABI et API 245
12.2.3.1. Choix de l'ABI 245
12.2.3.2. Définition de l'API 246
12.2.3.2.1. Simplicité d'utilisation 246
12.2.3.2.2. Interfaces synchrones et asynchrones 247
12.2.3.2.3. Gestion des allocations mémoire 247
12.2.3.2.4. Allocation des valeurs de retour 248
12.3. Considérations système 249
12.3.1. La sécurité 249
12.3.2. Le multithreading 250
12.3.2.1. Généralités 250
12.3.2.2. Utilisation du multithreading 251
12.3.2.3. Les pièges du multithreading 252
12.3.2.4. Multithreading et programmation objet 253
12.3.2.5. Limitations et contraintes liées au multithreading 254
12.3.3. Les signaux 255
12.3.4. Les bibliothèques de liaison dynamique 256
12.3.4.1. Les avantages des bibliothèques de liaison dynamique 256
12.3.4.2. Les mécanismes de chargement 256
12.3.4.3. Relogement et code indépendant de la position 257
12.3.4.4. Optimisation des bibliothèques de liaison dynamique 258
12.3.4.5. Initialisation des bibliothèques de liaison dynamique 261
12.3.5. Les communications réseau 262
12.3.5.1. Fiabilité des informations 262
12.3.5.2. Performances des communications 263
12.3.5.3. Pertes de connexion 264
II. La bibliothèque standard C++ 265
13. Services et notions de base de la bibliothèque standard 267
13.1. Encapsulation de la bibliothèque C standard 267
13.2. Définition des exceptions standards 269
13.3. Abstraction des types de données : les traits 272
13.4. Abstraction des pointeurs : les itérateurs 274
13.4.1. Notions de base et définition 274
13.4.2. Classification des itérateurs 275
13.4.3. Itérateurs adaptateurs 277
13.4.3.1. Adaptateurs pour les flux d'entrée / sortie standards 278
13.4.3.2. Adaptateurs pour l'insertion d'éléments dans les conteneurs 280
13.4.3.3. Itérateur inverse pour les itérateurs bidirectionnels 283
13.5. Abstraction des fonctions : les foncteurs 285
13.5.1. Foncteurs prédéfinis 285

13.5.2. Prédicats et foncteurs d'opérateurs logiques 290
13.5.3. Foncteurs réducteurs 291
13.6. Gestion personnalisée de la mémoire : les allocateurs 293
13.7. Notion de complexité algorithmique 297
13.7.1. Généralités 297
13.7.2. Notions mathématiques de base et définition 298
13.7.3. Interprétation pratique de la complexité 299
14. Les types complémentaires 301
14.1. Les chaînes de caractères 301
14.1.1. Construction et initialisation d'une chaîne 305
14.1.2. Accès aux propriétés d'une chaîne 306
14.1.3. Modification de la taille des chaînes 307
14.1.4. Accès aux données de la chaîne de caractères 308
14.1.5. Opérations sur les chaînes 310
14.1.5.1. Affectation et concaténation de chaînes de caractères 310
14.1.5.2. Extraction de données d'une chaîne de caractères 312
14.1.5.3. Insertion et suppression de caractères dans une chaîne 313
14.1.5.4. Remplacements de caractères d'une chaîne 314
14.1.6. Comparaison de chaînes de caractères 316
14.1.7. Recherche dans les chaînes 317
14.1.8. Fonctions d'entrée / sortie des chaînes de caractères 319
14.2. Les types utilitaires 320
14.2.1. Les pointeurs auto 320
14.2.2. Les paires 323
14.3. Les types numériques 324
14.3.1. Les complexes 325
14.3.1.1. Définition et principales propriétés des nombres complexes 325
14.3.1.2. La classe complex 327
14.3.2. Les tableaux de valeurs 330
14.3.2.1. Fonctionnalités de base des valarray 331
14.3.2.2. Sélection multiple des éléments d'un valarray 335
14.3.2.2.1. Sélection par un masque 335
14.3.2.2.2. Sélection par indexation explicite 336
14.3.2.2.3. Sélection par indexation implicite 337
14.3.2.2.4. Opérations réalisables sur les sélections multiples 339
14.3.3. Les champs de bits 340
15. Les flux d'entrée / sortie 345
15.1. Notions de base et présentation générale 345
15.2. Les tampons 347
15.2.1. Généralités sur les tampons 347
15.2.2. La classe basic_streambuf 348
15.2.3. Les classes de tampons basic_stringbuf et basic_filebuf 353
15.2.3.1. La classe basic_stringbuf 354
15.2.3.2. La classe basic_filebuf 356
15.3. Les classes de base des flux : ios_base et basic_ios 357
15.3.1. La classe ios_base 358
15.3.2. La classe basic_ios 364
15.4. Les flux d'entrée / sortie 367
15.4.1. La classe de base basic_ostream 367
15.4.2. La classe de base basic_istream 373
15.4.3. La classe basic_iostream 379
15.5. Les flux d'entrée / sortie sur chaînes de caractères 380

15.6. Les flux d'entrée / sortie sur fichiers 381
16. Les locales 385
16.1. Notions de base et principe de fonctionnement des facettes 386
16.2. Les facettes standards 391
16.2.1. Généralités 391
16.2.2. Les facettes de manipulation des caractères 392
16.2.2.1. La facette ctype 392
16.2.2.2. La facette codecvt 396
16.2.3. Les facettes de comparaison de chaînes 400
16.2.4. Les facettes de gestion des nombres 403
16.2.4.1. La facette num_punct 403
16.2.4.2. La facette d'écriture des nombres 405
16.2.4.3. La facette de lecture des nombres 406
16.2.5. Les facettes de gestion des monnaies 407
16.2.5.1. La facette money_punct 408
16.2.5.2. Les facettes de lecture et d'écriture des montants 410
16.2.6. Les facettes de gestion du temps 411
16.2.6.1. La facette d'écriture des dates 413
16.2.6.2. La facette de lecture des dates 413
16.2.7. Les facettes de gestion des messages 415
16.3. Personnalisation des mécanismes de localisation 417
16.3.1. Création et intégration d'une nouvelle facette 417
16.3.2. Remplacement d'une facette existante 421
17. Les conteneurs 425
17.1. Fonctionnalités générales des conteneurs 425
17.1.1. Définition des itérateurs 426
17.1.2. Définition des types de données relatifs aux objets contenus 427
17.1.3. Spécification de l'allocateur mémoire à utiliser 427
17.1.4. Opérateurs de comparaison des conteneurs 428
17.1.5. Méthodes d'intérêt général 429
17.2. Les séquences 429
17.2.1. Fonctionnalités communes 429
17.2.1.1. Construction et initialisation 429
17.2.1.2. Ajout et suppression d'éléments 431
17.2.2. Les différents types de séquences 432
17.2.2.1. Les listes 433
17.2.2.2. Les vecteurs 436
17.2.2.3. Les deques 438
17.2.2.4. Les adaptateurs de séquences 439
17.2.2.4.1. Les piles 439
17.2.2.4.2. Les files 440
17.2.2.4.3. Les files de priorités 440
17.3. Les conteneurs associatifs 442
17.3.1. Généralités et propriétés de base des clefs 443
17.3.2. Construction et initialisation 444
17.3.3. Ajout et suppression d'éléments 445
17.3.4. Fonctions de recherche 447
18. Les algorithmes 453
18.1. Opérations générales de manipulation des données 453
18.1.1. Opérations d'initialisation et de remplissage 454
18.1.2. Opérations de copie 455
18.1.3. Opérations d'échange d'éléments 456

18.1.4. Opérations de suppression d'éléments 457
18.1.5. Opérations de remplacement 459
18.1.6. Réorganisation de séquences 460
18.1.6.1. Opérations de rotation et de permutation 461
18.1.6.2. Opérations d'inversion 462
18.1.6.3. Opérations de mélange 463
18.1.7. Algorithmes d'itération et de transformation 464
18.2. Opérations de recherche 469
18.2.1. Opération de recherche d'éléments 469
18.2.2. Opérations de recherche de motifs 471
18.3. Opérations d'ordonnancement 473
18.3.1. Opérations de gestion des tas 474
18.3.2. Opérations de tri 476
18.3.3. Opérations de recherche binaire 480
18.4. Opérations de comparaison 483
18.5. Opérations ensemblistes 485
18.5.1. Opérations d'inclusion 485
18.5.2. Opérations d'intersection 486
18.5.3. Opérations d'union et de fusion 488
18.5.4. Opérations de différence 490
18.5.5. Opérations de partitionnement 492
19. Conclusion 495
A. Liste des mots clés du C/C++ 497
B. Priorités des opérateurs 499
C. Draft Papers 501
BIBLIOGRAPHIE 503

Liste des tableaux

1-1. Types de base du langage 10
1-2. Opérateurs du langage C/C++ 16
1-3. Chaînes de format de `printf` pour les types de base 29
1-4. Options de taille pour les types dérivés 30
1-5. Options d'alignements et de remplissange 31
2-1. Opérateurs de comparaison 35
2-2. Opérateurs logiques 35
3-1. Classes de stockages 57
3-2. Qualificatifs de constance 58
5-1. Trigraphes 96
7-1. Droits d'accès sur les membres hérités 114
12-1. Préfixes en notation hongroise simplifiée 220
14-1. Fonctions de recherche dans les chaînes de caractères 317
14-2. Fonctions spécifiques aux complexes 329
15-1. Options de formatage des flux 360
15-2. Modes d'ouverture des fichiers 361
15-3. Directions de déplacement dans un fichier 361
15-4. États des flux d'entrée / sortie 362
15-5. Manipulateurs des flux de sortie 371
15-6. Manipulateurs utilisant des paramètres 372
15-7. Manipulateurs des flux d'entrée 379
16-1. Fonctions C de gestion des dates 412
17-1. Méthodes spécifiques aux listes 434
A-1. Mots clés du langange 497
B-1. Opérateurs du langage 499

Avant-propos

Ce livre est un cours de C et de C++. Il s'adresse aux personnes qui ont déjà quelques notions de programmation dans un langage quelconque. Les connaissances requises ne sont pas très élevées cependant : il n'est pas nécessaire d'avoir fait de grands programmes pour lire ce document. Il suffit d'avoir vu ce qu'est un programme et compris les grands principes de la programmation.

Ce livre est structuré en deux grandes parties, traitant chacune un des aspects du C++. La première partie, contenant les chapitres 1 à 12, traite du langage C++ lui-même, de sa syntaxe et de ses principales fonctionnalités. La deuxième partie quant à elle se concentre sur la bibliothèque standard C++, qui fournit un ensemble de fonctionnalités cohérentes et réutilisables par tous les programmeurs. La bibliothèque standard C++ a également l'avantage d'utiliser les constructions les plus avancées du langage, et illustre donc parfaitement les notions qui auront été abordées dans la première partie. La description de la bibliothèque standard s'étend du chapitre 13 au chapitre 18.

Le plan de ce document a été conçu pour être didactique. Toutefois, certaines notions font référence à des chapitres ultérieurs. Cela n'est le cas que pour des points de détails, et les paragraphes en question sont identifiés en tant que tels. Ils pourront être passés en première lecture.

Si la bibliothèque standard C++ est décrite en détail, il n'en est pas de même pour les fonctions de la bibliothèque C. Vous ne trouverez donc pas dans ce livre la description des fonctions classiques du C, ni celle des fonctions les plus courantes de la norme POSIX. En effet, bien que présentes sur quasiment tous les systèmes d'exploitation, ces fonctions sont spécifiques à la norme POSIX et n'appartiennent pas au langage en soi. Seules les fonctions incontournables de la bibliothèque C seront donc présentées ici. Si vous désirez plus de renseignements, reportez-vous aux spécifications des appels systèmes POSIX de l'OpenGroup (http://www.unix-systems.org/single_unix_specification/), ou à la documentation des environnements de développement et à l'aide des kits de développement des systèmes d'exploitation (SDK).

Ce livre a pour but de présenter le langage C++ tel qu'il est décrit par la norme ISO 14882 du langage C++. Cette norme n'est pas disponible librement, aussi pourrez-vous vous rabattre sur les documents non officiels du projet de normalisation du langage les plus récents. Il s'agit des « Working Paper for Draft Proposed International Standard for Information Systems -- Programming Language C++ (http://casteyde.christian.free.fr/cpp/cours/drafts/index.html) », qui, bien qu'ils datent du 2 décembre 1996, sont encore tout à fait exploitables.

Notez que les compilateurs qui respectent cette norme se comptent encore sur les doigts d'une main, et que les informations et exemples donnés ici peuvent ne pas s'avérer exacts avec certains produits. En particulier, certains exemples ne compileront pas avec les compilateurs les plus mauvais. Notez également que certaines constructions du langage n'ont pas la même signification avec tous les compilateurs, parce qu'elles ont été implémentées avant que la norme ne les spécifie complètement. Ces différences peuvent conduire à du code non portable, et ont été signalées à chaque fois dans une note. Le fait que les exemples de ce livre ne fonctionnent pas avec de tels compilateurs ne peut donc pas être considéré comme une erreur, mais plutôt comme une non-conformité des outils utilisés, qui sera sans doute levée dans les versions ultérieures de ces produits.

Enfin, ce livre est un document vivant. Vous en trouverez toujours la dernière version sur mon site web (http://casteyde.christian.free.fr). Bien entendu, toute remarque est la bienvenue, et je tâcherai de corriger les erreurs que l'on me signalera et d'apporter les modifications nécessaires si un point est obscur. Si vous prenez le temps de m'envoyer les remarques et les erreurs que vous avez pu détecter, je vous saurais gré de vérifier au préalable qu'elles sont toujours d'actualité dans la dernière version de ce document. À cette fin, un historique des révisions a été inclus en première page pour permettre l'identification des différentes éditions de ce document.

I. Le langage C++

Tout le début de cette partie (chapitres 1 à 6) traite des fonctionnalités communes au C et au C++, en insistant bien sur les différences entre ces deux langages. Ces chapitres présentent essentiellement la syntaxe des constructions de base du C et du C++. Le début de cette partie peut donc également être considéré comme un cours allégé sur le langage C. Cependant, les constructions syntaxiques utilisées sont écrites de telle sorte qu'elles sont compilables en C++. Cela signifie qu'elles n'utilisent pas certaines fonctionnalités douteuses du C. Ceux qui désirent utiliser la première partie comme un cours de C doivent donc savoir qu'il s'agit d'une version épurée de ce langage. En particulier, les appels de fonctions non déclarées ou les appels de fonctions avec trop de paramètres ne sont pas considérés comme des pratiques de programmation valables.

Les chapitres suivants (chapitres 7 à 11) ne traitent que du C++. Les constructions utilisées pour permettre la programmation orientée objet, ainsi que toutes les extensions qui ont été apportées au langage C pour gérer les objets y sont décrits. Le mécanisme des exceptions du langage, qui permet de gérer les erreurs plus facilement, est ensuite présenté, de même que les mécanismes d'identification dynamique des types. Enfin, les notions d'espaces de nommage et de modèles de fonctions et de classes seront décrites. Ces dernières fonctionnalités sont utilisées intensivement dans la bibliothèque standard C++, aussi la lecture complète de la première partie est-elle indispensable avant de s'attaquer à la deuxième.

Enfin, le dernier chapitre de cette partie (chapitre 12) traite des règles de codage et donne des conseils utiles à la réalisation de programmes plus sûrs, plus maintenables et plus évolutifs. La lecture de ce chapitre n'est donc pas liée au langage C/C++ et n'est pas techniquement obligatoire, mais elle donne des idées qui pourront être suivies ou adaptées dans le but d'acquérir dès le départ de bonnes habitudes.

Dans toute cette première partie, la syntaxe sera donnée, sauf exception, avec la convention suivante : ce qui est entre crochets ('`[` et '`]`') est facultatif. De plus, quand plusieurs éléments de syntaxe sont séparés par une barre verticale ('`|`'), l'un de ces éléments, et un seulement, doit être présent (c'est un « ou » exclusif). Enfin, les points de suspension désigneront une itération éventuelle du motif précédent.

Par exemple, si la syntaxe d'une commande est la suivante :

```
[fac|rty|sss] zer[(kfl[,kfl[...]])];
```

les combinaisons suivantes seront syntaxiquement correctes :

```
zer;
fac zer;
rty zer;
zer(kfl);
sss zer(kfl,kfl,kfl,kfl);
```

mais la combinaison suivante sera incorrecte :

```
fac sss zer()
```

pour les raisons suivantes :

- *fac* et *sss* sont mutuellement exclusifs, bien que facultatifs tous les deux ;
- au moins un *kfl* est nécessaire si les parenthèses sont mises ;
- il manque le point virgule final.

Rassurez-vous, il n'y aura pratiquement jamais de syntaxe aussi compliquée. Je suis sincèrement désolé de la complexité de cet exemple.

Chapitre 1. Première approche du C/C++

L'apprentissage d'un langage de programmation n'est pas chose aisée. Une approche progressive est nécessaire. Ce chapitre donnera donc les concepts de base de la programmation et présentera le langage C/C++ de manière pratique, à l'aide d'exemples de complexité progressive. Les notions plus complexes seront présentées dans les chapitres ultérieurs, une fois que l'on aura éliminé les premières appréhensions.

1.1. Les ordinateurs, les langages et le C++

1.1.1. Les ordinateurs et la programmation

Les ordinateurs sont des machines conçues pour être génériques et effectuer des opérations a priori non prévues lors de leur conception. Ils se distinguent en cela des machines conçues dans un but spécifique, afin de réaliser une tâche prédéterminée.

C'est cette généricité qui leur donne toute leur utilité. Si, dans certaines situations, des machines spécifiques sont particulièrement appropriées, notamment pour des raisons de performances, les ordinateurs sont incontournables dès lors que les tâches à accomplir sont diverses ou dès lors qu'elles sont nouvelles et n'ont pas de solution existante. C'est pour cette raison qu'ils sont particulièrement utilisés dans le monde de la simulation, où les développements spécifiques sont réalisés afin d'étudier le phénomène à observer. De même, ils sont utilisés pour modéliser et concevoir les machines spécifiques qui effectueront la tâche au final.

Malgré leur généricité, les ordinateurs sont parvenus à des performances plus qu'honorables en termes de puissance de calcul, de stockage et de communication, et peuvent à présent être utilisés là où des circuits spécifiques étaient nécessaires il y a encore peu de temps. Le gain est alors double : les produits sont évolutifs et les coûts de fabrication moindre, puisque le marché de chaque circuit spécifique est bien plus restreint que celui des circuits génériques à présent.

Mais, en raison même de leur généricité, ces machines ne savent effectuer que très peu de choses par défaut. En réalité, elles ne savent exécuter que des instructions de base. Par conséquent, pour parvenir au but recherché, il est nécessaire de spécifier de longues listes d'instructions, devant être exécutées en séquence, voire parfois en parallèle, selon l'architecture matérielle de l'ordinateur. Cette phase de spécification s'appelle la *programmation*.

1.1.2. Les langages de programmation

La programmation directe d'un ordinateur n'est pas une tâche aisée, car les instructions élémentaires ne font réellement pas grand chose chacune. Afin de faciliter cette programmation, des logiciels complets ont été écrits. Les plus importants sont sans doute les systèmes d'exploitation, qui prennent en charge la gestion de l'ordinateur lui-même et fournissent des fonctionnalités de haut niveau, et les logiciels de langages de programmation, dont le rôle est de permettre de réaliser d'autres programmes à l'aide d'un langage de plus haut niveau que le langage de la machine cible. À présent, les systèmes et les logiciels des langages de programmation sont eux-mêmes programmés à l'aide des langages de programmation, si bien que la programmation directe des ordinateurs en langage machine n'est plus réservée que pour des applications très spécifiques ou lorsque de très grandes performances sont recherchées.

Les programmes écrits dans un langage de programmation ne sont évidemment pas compréhensibles par l'ordinateur. Les logiciels de langages de programmation doivent donc s'assurer que les programmes sont correctement exécutés par l'ordinateur. Deux solutions ont été adoptées pour résoudre ce problème : soit le texte du programme (c'est-à-dire ce que l'on appelle le « code source ») est traduit en langage machine par un logiciel appelé « compilateur », puis exécuté nativement, soit il est lu et interprété par un logiciel qui effectue les tâches programmées lui-même. Les langages de la première catégorie sont des langages dits « compilés », alors que les langages de la deuxième catégorie sont des langages dits « interprétés ». Les deux techniques ont leurs avantages et leurs inconvénients. Les langages compilés sont les plus rapides, alors que les langages interprétés sont les plus facile à mettre au point (le programme pouvant souvent être débogué et modifié en cours d'exécution par l'interpréteur).

Les langages de programmation permettent, au sens large, d'exprimer ce que l'on veut faire faire à un ordinateur. Cela implique que, contrairement aux langages naturels tels que le français ou l'anglais, toute ambiguïté doit être absente. Cela suppose une syntaxe définie mathématiquement, et une sémantique parfaitement définie sur les contructions du langage que l'on s'autorise à utiliser. Les langages de programmation sont donc définis de manière rigoureuse, et rejoignent les mathématiques. Toutefois, en pratique, ces langages ont pour but d'être humainement compréhensibles et utilisables. Les aspects théoriques ne sont donc intéressants qu'au niveau de la recherche, et si le langage répond au besoin initial (qui est de programmer l'ordinateur), il doit être facile à utiliser.

La théorie des langages de programmation distingue essentiellement trois grandes classes de langages de programmation, dont les langages sont plus ou moins appropriés aux différents problèmes que le programmeur doit résoudre. Ces trois classes de langages sont respectivement :

- les langages impératifs, qui permettent de faire des programmes constitués de suites d'instructions permettant de modifier l'état de l'ordinateur stocké dans sa mémoire ;
- les langages fonctionnels, qui n'utilisent pas la notion d'état mais qui considèrent les programmes comme des suites de fonctions dont les résultats constituent le comportement du programme exécuté ;
- les langages logiques, qui s'intéressent plus aux résultats que le programme doit fournir, et cherchent à les caractériser par des contraintes logiques que l'ordinateur résoud ensuite grâce à un moteur d'inférence.

Bien entendu, chaque type de langage a ses avantages et ses inconvénients, et donc ses domaines d'applications. Le principe fondamental est donc ici de choisir le type de langage en fonction du problème, afin de le résoudre le plus facilement. Par exemple, les langages logiques sont particulièrement appréciés pour réaliser les systèmes experts, dont le rôle est de fournir une solution à un problème à partir de règles définies par les hommes de l'art du domaine considéré. Le programme se base dans ce cas sur ces règles et les combine logiquement pour obtenir la solution du problème (la manière de l'obtenir important ici moins que les règles que ce résultat doit vérifier).

L'avantage des langages impératifs, dont fait partie le C/C++, réside dans le fait que les ordinateurs sont des machines elles-mêmes conçues pour exécuter des programmes impératifs (c'est-à-dire les listes d'instructions du langage machine). Les langages impératifs sont donc très faciles à compiler, et permettent d'accéder aux fonctionnalités des ordinateurs de manière extrêmement simple. Bien entendu, les performances sont garanties, si bien sûr les algorithmes utilisés sont appropriés. Enfin, le modèle impératif correspond particulièrement bien aux réalités industrielles et aux automates, avec lesquels la communication se fait généralement sous forme d'ordres. En revanche, contrairement aux autres classes de langage, les langages impératifs sont sujets à des bogues provenant d'incohérences dans la gestion de l'état du programme. Les techniques de développement permettent toutefois de limiter ces erreurs, comme nous le verrons plus loin dans ce document.

1.1.3. Le langage C/C++

Le C/C++ est un langage impératif compilé, du même type que le Pascal par exemple. C'est l'un des langages de programmation les plus utilisés actuellement. Il est à la fois facile à utiliser et très efficace. Il souffre cependant de la réputation d'être compliqué, illisible et, en raison de son aspect bas niveau, de permettre des bogues dans la gestion de la mémoire.

Cette réputation est en partie justifiée. La complexité du langage est inévitable lorsqu'on cherche à avoir beaucoup de fonctionnalités. La lisibilité des programmes en revanche ne dépend que de la bonne volonté du programmeur, et peut être améliorée à l'aide de conventions de codage simples. Quant aux bogues bas niveau que le langage permet, ils sont la contrepartie d'un contrôle total du comportement du programme et donc de la machine. De plus, ils deviennent relativement rares avec l'expérience et peuvent également être grandement limités par l'utilisation de règles de codage simples.

Les caractéristiques du C/C++ en font un langage idéal pour certains types de projets. Il est incontournable dans la réalisation des programmes orientés systèmes ou temps réels. Il est également très utilisé par de grands programmes, tant en raison de l'historique que des performances pratiques que l'on obtient au final. Ces performances sont garanties aussi bien par le fait que le programmeur maîtrise réellement le programme, par la possibilité d'accéder à l'ensemble des fonctions système, et par les optimisations fournies par les compilateurs actuels. C'est également un langage normalisé et présent sur l'ensemble des plateformes, ce qui permet de réaliser des programmes portables au niveau source (les langages interprétés prétendent une portabilité supérieure, mais celle-ci reste relative au contexte d'exécution et déplace le problème vers la disponibilité des interpréteurs sur l'ensemble des plateformes). Enfin, les outils disponibles sont nombreux et fiables.

Le langage C++ constitue en soi une extension du langage C qui apporte de réels avantages :

- contrôle d'erreurs accru grâce à un typage fort des données ;
- facilité d'utilisation des langages objets ;
- grand nombre de fonctionnalités complémentaires ;
- performances du C ;
- facilité de conversion des programmes C en C++, et, en particulier, possibilité d'utiliser toutes les fonctionnalités du langage C.

On dispose donc de quasiment tout : puissance, fonctionnalité, portabilité et sûreté. La richesse du contrôle d'erreurs du langage, basé sur un typage très fort, permet de signaler un grand nombre d'erreurs à la compilation. Toutes ces erreurs sont autant d'erreurs que le programme ne fait pas à l'exécution. Le C++ peut donc être considéré comme un « super C ». Le revers de la médaille est que les programmes C ne se compilent pas directement en C++ : il est courant que de simples avertissements en C soient des erreurs bloquantes en C++. Quelques adaptations sont souvent nécessaires. Cependant, celles-ci sont minimes, puisque la syntaxe du C++ est basée sur celle du C. On remarquera que tous les programmes C peuvent être corrigés pour compiler à la fois en C et en C++.

1.1.4. Les outils de programmation

Quel que soit le type d'ordinateur et le type de système d'exploitation que vous utilisez, il existe certainement un compilateur C/C++ pour cette plateforme. Pour certains systèmes, le compilateur est un compilateur dit « croisé », c'est-à-dire un compilateur qui produit du code exécutable pour une autre machine que celle sur laquelle il fonctionne. Sur les plateformes les plus courantes, un

choix abondant d'environnements de développement est proposé par les éditeurs de logiciels ou par la communauté des logiciels libres. Ces produits sont généralement de qualité, et ils sont à présent tous utilisables pour développer des applications C/C++, et souvent même dans d'autres langages.

Si vous disposez de ces environnement, je vous invite à les utiliser. Dans le cas contraire, vous devrez installer au minimum un compilateur. Le compilateur GCC de la Free Software Foundation sera généralement un bon choix, car il s'agit du compilateur installé par défaut sur les machines fonctionnant sous Linux et sous MacOS X (voir le site de GCC (http://gcc.gnu.org)), et il existe une version installable facilement pour Windows (disponible sur le site du projet MinGW (http://www.mingw.org), Minimalist GNU for Windows). Vous devriez également vous assurer que vous disposez du débogueur en ligne de commande GDB, qui est le débogueur par défaut sous Linux et MacOS X, et qui est également fourni par le projet MinGW. La suite de ce document supposera que vous utilisez ces logiciels, en raison de leur qualité et de leur disponibilité sur l'ensemble des plateformes.

Vous pourrez vous assurer que GCC est correctement installé en exécutant la commande suivante dans une fenêtre de commande (émulateur de terminal sous Linux ou MacOS X, ou fenêtre MS-DOS sous Windows) :

```
c++ --version
```

Cette commande doit afficher un résultat semblable au suivant :

```
c++ (GCC) 3.4.4
Copyright (C) 2004 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

La première ligne affichée indique la version du logiciel.

> **Note :** La commande **c++** est la commande classiquement utilisée sous Unix pour appeler le compilateur pour le langage C++. Si l'on désire ne faire que des programmes C, la commande à utiliser est **cc**.
>
> Ces commandes sont généralement des alias pour les commandes natives de GCC, qui sont **gcc** et **g++** respectivement pour les langages C et C++. Si ces alias ne sont pas définis, vous pouvez les définir ou utiliser les commandes natives de GCC directement.

1.2. Notre premier programme

Nous allons maintenant entrer dans le vif du sujet et présenter notre premier programme. L'usage en informatique est de faire un programme qui n'affiche qu'une ligne de texte et salue la compagnie : l'inévitable « Hello World! »...

1.2.1. Hello World!

Voici donc notre premier programme :

Exemple 1-1. Hello World!

```
#include <stdlib.h>
#include <stdio.h>
```

```
int main(void)
{
    printf("Hello World!\n");
    return EXIT_SUCCESS;
}
```

Ce programme se compile avec la commande suivante :

```
c++ hello.cpp -o hello.exe
```

en supposant que le fichier source se nomme `hello.cpp` et que le nom de l'exécutable à produire soit `hello.exe`. Son exécution *dans un émulateur de terminal ou une fenêtre de commandes* (pas à partir d'un environnement graphique intégré, qui lancera le programme et fermera sa fenêtre immédiatement après la fin de celui-ci) donne le résultat suivant :

```
Hello World!
```

> **Note :** L'extension des fichiers sources pour le langage C est classiquement « `.c` ». Les fichiers sources pour le langage C++ sont souvent « `.C` », « `.cc` » ou « `.cpp` ». L'extension « `.C` » est particulièrement déconseillée, puisqu'elle ne se distingue de l'extension des fichiers C que par la casse, et que certains systèmes de fichiers ne sont pas capables de distinguer des fichiers uniquement par leur casse. L'extension « `.cpp` » est reconnue par l'ensemble des compilateurs et est par conséquent fortement conseillée.
>
> De même, il n'est pas nécessaire d'utiliser l'extension « `.exe` » sous Linux et MacOS X. En revanche, c'est impératif sous Windows, parce que celui-ci se base sur les extensions pour déterminer la nature des fichiers. Par conséquent, elle a été ajoutée dans cet exemple.

1.2.2. Analyse du programme

Décorticons à présent ce programme. Les deux premières lignes permettent d'inclure le contenu de deux fichiers nommés `stdlib.h` et `stdio.h`. Ces fichiers, que l'on nomme généralement des fichiers d'*en-tête*, contiennent les déclarations de fonctions et de constantes que le reste du programme va utiliser. Ces déclarations sont nécessaires pour pouvoir les utiliser. Sans elles, le compilateur signalerait qu'il ne connaît pas ces fonctions et ces constantes. Comme vous pouvez le voir dans cet exemple, il est d'usage d'utiliser l'extension « `.h` » pour les fichiers d'en-tête.

Il existe de nombreux fichiers d'en-tête, chacun regroupant un jeu de fonctions utilitaires. Connaître les fonctionnalités disponibles et les fichiers d'en-tête à inclure pour y accéder n'est pas une tâche facile, et ce savoir ne peut être acquis que par l'expérience et la consultation de la documentation des environnements de développement. En particulier, un grand nombre de fonctionnalités sont fournies avec la bibliothèque de fonctions du langage C, mais les décrire toutes nécessiterait un livre complet en soi. En ce qui concerne notre exemple, sachez que le fichier d'en-tête `stdlib.h` contient les déclarations des fonctions et des constantes de base de la bibliothèque C, et que le fichier d'en-tête `stdio.h` les déclarations des principales fonctions d'entrée/sortie permettant de lire et interpréter des données et d'afficher les résultats du programme.

La quatrième ligne contient la déclaration de la fonction « `main` ». Cette fonction est le point d'entrée du programme, c'est-à-dire la fonction que le programme exécutera lorsqu'il sera lancé. Le système identifie cette fonction justement par le fait qu'elle s'appelle « `main` », ne vous amusez-donc pas à changer le nom de cette fonction. Pour information, « main » signifie « principal » en anglais, ce qui indique bien que cette fonction est la fonction principale du programme.

Comme vous pouvez le constater, le nom de la fonction est précédé du mot clé « int », qui représente le type de données entier (type de données utilisé pour stocker des données entières). Cela signifie que cette fonction retourne une valeur entière. En effet, il est d'usage que les programmes renvoient un code numérique de type entier au système à la fin de leur exécution. Généralement, la valeur 0 indique une exécution correcte, et toute autre valeur une erreur. En C/C++, cette valeur est tout simplement la valeur retournée par la fonction `main`.

Généralement, les arguments passés au programme par le système sur sa ligne de commandes peuvent être récupérés en tant que paramètres de la fonction `main`. En C/C++, les paramètres de fonctions sont spécifiés entre parenthèses, sous la forme d'une liste de paramètres séparés par des virgules. Dans cet exemple cependant, notre programme ne prend pas de paramètres, et nous ne désirons pas récupérer la ligne de commande utilisée par le système pour lancer le programme. C'est pour cela que le mot clé `void` est-il utilisé dans la liste des paramètres pour indiquer que celle-ci est vide.

Après la ligne de déclaration de la fonction `main`, vous trouverez son *implémentation* (c'est-à-dire le code source qui décrit ce qu'elle fait). Celle-ci est encadrée par des accolades (caractères '`{`' et '`}`'), qui sont les caractères utilisés pour regrouper des instructions.

Le corps de la fonction principale commence par appeler la fonction de la bibliothèque C « `printf` ». Cette fonction, dont la déclaration est dans le fichier d'en-tête `stdio.h`, permet d'afficher une chaîne de caractères sur la sortie standard du programme. Dans notre exemple, on se contente d'afficher la chaîne de caractères « `Hello World!` » à l'écran. Vous pouvez ainsi constater que les chaînes de caractères sont fournies entre guillemets anglais (caractère '`"`'). La fin de la chaîne de caractères affichée contient un marqueur de saut de ligne (« newline » en anglais), représenté par « `\n` ». Nous verrons plus loin que le C/C++ fournit un certain nombre de marqueurs spéciaux de ce type pour représenter les caractères de contrôle ou non imprimables.

> **Note :** La fonction `printf` est en réalité beaucoup plus puissante et permet d'afficher virtuellement n'importe quoi, simplement à partir des valeurs des informations à afficher et d'une chaîne de format (qui décrit justement comment ces informations doivent être affichées). Cette fonction permet donc d'effectuer des sorties formatées sur la sortie standard du programme, d'où son nom (« PRINT Formatted »).

Enfin, la dernière opération que fait la fonction `main` est de retourner le code de résultat du programme au système d'exploitation. En C/C++, la valeur de retour des fonctions est spécifiée à l'aide du mot clé « `return` », suivi de cette valeur. Dans cet exemple, nous utilisons la constante « `EXIT_SUCCESS` », définie dans le fichier d'en-tête `stdlib.h`. Cette constante a pour valeur la valeur utilisée pour signaler qu'un programme s'est exécuté correctement au système (0 sur la plupart des systèmes). Si nous avions voulu signaler une erreur, nous aurions par exemple utilisé la constante « `EXIT_FAILURE` ».

1.2.3. Généralisation

Tout programme écrit dans un langage impératif a pour but d'effectuer des opérations sur des données. La structure fondamentale d'un programme en C/C++ est donc la suivante :

```
                ENTRÉE DES DONNÉES
(clavier, souris, fichier, autres périphériques)
```

```
                    |
          TRAITEMENT DES DONNÉES
                    |
           SORTIE DES RÉSULTATS
(écran, imprimante, fichier, autres périphériques)
```

Pour les programmes les plus simples, les données proviennent du flux d'entrée standard, d'un fichier ou d'une connexion réseau, et les résultats sont émis sur le flux de sortie standard, dans un fichier ou vers une connexion réseau. Les flux d'entrée et de sortie standards sont les flux par défaut pour les programmes, et sont généralement associés au clavier et à la console du terminal ou de l'émulateur de terminal dans lequel le programme fonctionne. Dans l'exemple que nous venons de voir, il n'y a aucune entrée, mais une sortie qui est effectuée sur le flux de sortie standard grâce à la fonction `printf`.

Pour les programmes plus évolués en revanche, le traitement n'est pas aussi linéaire, et s'effectue souvent en boucle, de manière à répondre interactivement aux entrées. Par exemple, pour les programmes graphiques, les données sont reçues de la part du système sous forme de *messages* caractérisant les *événements* générés par l'utilisateur ou par le système lui-même (déplacement de souris, fermeture d'une fenêtre, appui sur une touche, etc.). Le traitement du programme est alors une boucle infinie (que l'on appelle la *boucle des messages*), qui permet de récupérer ces messages et de prendre les actions en conséquence. Dans ce cas, la sortie des données correspond au comportement que le programme adopte en réponse à ces messages. Cela peut être tout simplement d'afficher les données saisies, ou, plus généralement, d'appliquer une commande aux données en cours de manipulation.

Tous ces traitements peuvent être dispersés dans le programme. Par exemple, une partie du traitement peut avoir besoin de données complémentaires et chercher à les récupérer sur le flux d'entrée standard. De même, un traitement complexe peut être découpé en plusieurs sous-programmes, qui sont appelés par le programme principal.

Ces sous-programmes sont, dans le cas du C/C++, des fonctions ou des procédures (c'est-à-dire des fonctions qui ne retournent aucune valeur), qui peuvent être appelées par la fonction principale, et qui peuvent appeler eux-mêmes d'autres fonctions ou procédures. Bien entendu, des fonctions utilisateurs peuvent être définies. La bibliothèque C standard fournit également un grand nombre de fonctions utilitaires de base, et des bibliothèques complémentaires peuvent être utilisées pour des fonctionnalités complémentaires.

Les données manipulées par les programmes impératifs sont stockées dans des *variables*, c'est-à-dire des zones de la mémoire. C'est l'ensemble de ces variables qui constitue l'état du programme. Les variables sont modifiées par le traitement des données, dans le cadre d'opérations bien précises.

Comme la plupart des langages, le C/C++ utilise la notion de *type* afin de caractériser les opérations réalisables sur les données et la nature des données qu'elles représentent. Cela permet d'accroître la fiabilité des programmes, en évitant que les données soient manipulées de manière incompatible. Par exemple, on ne peut pas ajouter des pommes à des bananes, sauf à définir cette opération bien précisément. Les opérations effectuées sur les données dépendent donc de leur type.

Le langage C/C++ fournit des types de base et des opérations prédéfinies sur ces types. Les opérations qui peuvent être faites sont réalisées via l'application d'un *opérateur* sur les expressions auxquelles il s'applique. Par exemple, l'addition de deux entiers et leur affectation à une variable « `a` » s'écrit de la manière suivante :

```
a=2+3
```

Cette expression utilise l'opérateur d'addition entre entiers, et l'opérateur d'affectation d'un entier du type de la variable `a`.

Évidemment, le programmeur pourra définir ses propres types de données. Il n'est pas possible de définir de nouveaux opérateurs, mais en C++ (pas en C) les opérateurs peuvent être redéfinis pour les nouveaux types que l'utilisateur a créés. Si des besoins plus spécifiques se présentent, il faudra écrire des fonctions pour manipuler les données.

> **Note :** En réalité, les opérateurs constituent une facilité d'écriture, et ils sont équivalents à des fonctions prenant leurs opérandes en paramètres et retournant leur résultat. Logiquement parlant, il n'y a pas de différence, seule la syntaxe change. L'écriture précédente est donc strictement équivalente à :
>
> ```
> a=ajoute(2,3)
> ```

La suite de ce chapitre va présenter la manière de déclarer les variables ainsi que les types de base du langage, la manière d'écrire les instructions et d'utiliser les opérateurs, et la syntaxe utilisée pour définir de nouvelles fonctions. Les notions de flux d'entrée/sortie seront ensuite présentées afin de permettre la réception des données à traiter et de fournir en retour les résultats.

1.3. Les commentaires en C/C++

Les commentaires sont des portions de texte insérées dans le code source d'un programme et qui ne sont pas prises en compte par le compilateur. Ils permettent, comme leur nom l'indique, de documenter et d'expliquer en langage naturel tout ce que le programmeur pense être nécessaire pour la bonne compréhension du code source.

Les commentaires sont donc absolument nécessaires, mais ne doivent bien entendu ne pas être inutiles : trop de commentaires tue le commentaire, parce que les choses importantes sont dans ce cas noyées dans les banalités. Les exemples de ce document utiliseront bien entendu des commentaires dont le but sera d'appuyer les explications qui y sont relatives. Nous allons donc voir immédiatement comment réaliser un commentaire en C et en C++.

Il existe deux types de commentaires : les commentaires C et les commentaires C++. Les commentaires C permettent de mettre tout un bloc de texte en commentaire. Ils sont bien entendu disponibles en C++. Les commentaires C++ en revanche ne sont, normalement, pas disponibles en C, sauf extensions du compilateur C. Ils ne permettent de commenter que la fin d'une ligne, mais sont généralement plus pratiques à utiliser.

Les commentaires C commencent avec la séquence barre oblique - étoile (« `/*` »). Ils se terminent avec la séquence inverse : une étoile suivie d'une barre oblique (« `*/` »).

Exemple 1-2. Commentaire C

```
/*   Ceci est un commentaire C   */
```

Du fait que la fin du commentaire est parfaitement identifiée par une séquence de caractères, ils peuvent s'étendre sur plusieurs lignes.

Les commentaires C++ en revanche s'arrêtent à la fin de la ligne courante, et de ce fait n'ont pas de séquence de terminaison. Ils permettent de commenter plus facilement les actions effectuées sur la ligne courante, avant le commentaire, et forcent ainsi souvent le programmeur à plus de concision

dans ses commentaires (ce qui est une bonne chose). Les commentaires C++ commencent par la séquence constituée de deux barres obliques. Par exemple :

Exemple 1-3. Commentaire C++

```
ligne de code quelconque     // Ceci est un commentaire C++
ligne de code suivante
```

Les commentaires C++ peuvent être placés dans un commentaire C. Ceci est très pratique lorsque l'on veut supprimer toute un bloc de code source commenté avec des commentaires C++, pour tester une variante par exemple.

En revanche, les commentaires C ne peuvent pas être placés dans un commentaire de portée plus générale, parce qu'ils ne sont pas récursifs. En effet, lors de l'ajout d'un commentaire C, toute ouverture de commentaires C dans le bloc commenté est elle-même mise en commentaire. De ce fait, le commentaire s'arrête dès la première séquence étoile-barre oblique rencontrée, et la séquence de terminaison du commentaire de portée globale provoque une erreur de compilation.

Par exemple, dans le code suivant, la deuxième séquence « `*/` » est analysée par le compilateur, qui y voit une erreur de syntaxe (les commentaires C++ sont utilisés pour expliquer l'exemple) :

```
/* Début de commentaire englobant...

// Ligne commentée, avec son commentaire original :
a = 2+3; /* On affecte 5 à a */  // À partir d'ici, le commentaire C est fini !

// Fin supposée du commentaire englobant, interprété
// comme les opérateurs de multiplication et de division :
*/
```

Ce problème peut-être génant, et on verra à l'usage que la possibilité de commenter un commentaire C++ par un commentaire C incite fortement à utiliser les commentaires C++ en priorité.

1.4. Les variables

Les *variables* sont des représentations de zones de la mémoire utilisée par le programme pour stocker son état. Elles sont essentielles, puisqu'elles contiennent les données sur lesquelles les programmes travaillent.

1.4.1. Définition des variables

Toute variable doit avoir un nom qui l'identifie (on appelle un tel nom un « *identificateur* »), grâce auquel elle pourra être manipulée dans la suite du programme. Comme nous l'avons dit plus haut, les variables C/C++ disposent également d'un type, ce qui signifie que des informations relatives à la nature de la variable sont associées à ce nom par le compilateur, afin que celui-ci puisse déterminer les opérations qui peuvent être appliquées à la variable.

La *déclaration* d'une variable permet donc d'associer un identificateur à cette variable et d'indiquer au compilateur son type. En pratique, ces deux opérations sont réalisées en même temps que la réservation de la mémoire nécessaire au stockage de la variable, lors de sa définition. La *définition* d'une variable est donc l'opération qui permet à la fois de la déclarer et de lui attribuer une zone mémoire.

Nous verrons la différence importante entre la déclaration et la définition lorsque nous réaliserons des programmes dont le code source est réparti dans plusieurs fichiers.

Les noms d'identificateurs doivent être choisis conformément aux règles du langage. Le C++ est relativement permissif quant aux caractères utilisables dans les noms d'identificateurs, mais en pratique les compilateurs ne le sont pas. Par conséquent, on se restreindra aux caractères alphanumériques non accentués et au caractère de soulignement bas (caractère '_'), sachant que les chiffres ne peuvent pas être utilisés au début d'un nom. De plus, un certain nombre de noms sont réservés et ne peuvent être utilisés (mots clés et quelques constantes ou noms réservés par le langage). Vous trouverez la liste des mots clés dans l'Annexe A.

Par exemple, les noms « `couleur_objet` », « `AgeBestiole` » et « `Parametre5` » sont acceptables, mais pas « `6tron` », « `résultat` », « `int` » ou « `EXIT_SUCCESS` ».

La syntaxe utilisée pour définir une variable simple est la suivante :

```
type identificateur;
```

où type est le type de la variable et `identificateur` est son nom. Il est possible de créer et d'initialiser une série de variables dès leur création avec la syntaxe suivante :

```
type identificateur[=valeur][, identificateur[=valeur][...]];
```

Exemple 1-4. Définition de variables

```
int i=0, j=0;     /* Définit et initialise deux entiers à 0 */
int somme;        /* Déclare une autre variable entière */
```

En C, les variables peuvent définies en dehors de toute fonction (variables dites *globales*), au début des fonctions (variables *locales à la fonction*), ou au début des blocs d'instructions. Le C++ est plus souple et autorise généralement la définition de variables également au sein d'un bloc d'instructions. Cela permet de ne définir une variable temporaire que là où l'on en a besoin, donc de réduire la globalité de ces variables et de minimiser les erreurs de programmation dues aux effets de bords. De plus, cela permet d'éviter d'avoir à connaître les variables temporaires nécessaires à l'écriture du morceau de code qui suit leur définition, et accroît la lisibilité des programmes.

La définition d'une variable ne suffit pas, en général, à l'initialiser. Les variables non initialisées contenant des valeurs aléatoires, il faut éviter de les utiliser avant une initialisation correcte. Initialiser les variables que l'on déclare à leur valeur par défaut est donc une bonne habitude à prendre. L'initialisation est d'ailleurs obligatoire pour les variables « constantes » que l'on peut déclarer avec le mot clé `const`, car ces variables ne peuvent pas être modifiées après leur définition. Ce mot clé sera présenté en détail ultérieurement, dans la Section 3.6.

1.4.2. Les types de base du C/C++

Nous avons déjà vu le type de données int dans notre premier programme pour la valeur de retour de la fonction `main`, et dans l'exemple de déclaration de variables précédent. Le langage C/C++ fournit bien entendu d'autres types de données, ainsi que la possibilité de définir ses propres types. Nous verrons la manière de procéder pour cela dans le Section 3.2.

En attendant, voyons les types de données les plus simples du langage :

Tableau 1-1. Types de base du langage

Type	Description
void	Le type vide. Ce type est utilisé pour spécifier le fait qu'il n'y a pas de valeur possible pour l'entité de ce type Cela a une utilité, entre autres, pour faire des procédures (fonctions ne renvoyant aucune valeur).
bool	Type des valeurs booléennes. Ce type est utilisé pour représenter la véracité d'une expression. Il ne peut valoir que `true` (expression vraie) ou `false` (expression fausse). Ce type de données n'est disponible qu'en C++.
char	Type des caractères.
wchar_t	Type des caractères étendus. Utilisé pour représenter, généralement en Unicode, les caractères qui ne sont pas représentables dans un jeu de caractères 8 bits.
int	Type de base des entiers signés.
float	Type de base des nombres en virgule flottante.
double	Type des nombres en virgule flottante en double précision.

Les types de données bool, char, wchar_t et int peuvent être utilisés pour stocker des valeurs entières (le cas du type bool étant dégénéré, puisque seules deux valeurs sont utilisables !). De ce fait, on dit que ce sont des *types intégraux*. La conversion d'un entier en booléen se fait avec la règle selon laquelle 0 est faux et toute valeur non nulle est vraie. Inversement, `false` est considéré comme valant 0 et `true` comme valant 1.

Les types de données float et double sont généralement utilisés pour les calculs numériques. Ils permettent de représenter des nombres réels, sous la forme de leurs chiffres les plus significatifs et d'un exposant exprimant la position de l'unité par rapport à ces chiffres (par exemple « 3,14159 » ou « 1,535x10^12 ». De ce fait, cet exposant indique la position de la virgule du nombre réel représenté par rapport aux chiffres significatifs. Comme cette position dépend de cet exposant, cette position n'est pas fixée par le type de données en soi. C'est la raison pour laquelle on appelle ces nombres des nombres en *virgule flottante*.

> **Note :** Les types flottants sont très pratiques pour les calculs numériques, mais ils souffrent de problèmes d'arrondis et d'erreurs de représentation et de conversion très gênants. En particulier, les nombres flottants ne permettent pas de stocker les nombres dont la partie décimale est infinie (comme 1/3 par exemple). En effet, l'ordinateur est une machine finie, et il ne peut stocker une infinité de nombres après la virgule. Il y a donc nécessairement une *erreur de représentation* du type de données utilisé pour les nombres réels, quel que soit le type utilisé ! De ces erreurs de représentation découlent les erreurs d'arrondi, qui font que deux nombres supposés égaux ne le seront qu'à un epsilon près relativement faible. Les nombres flottants sont donc particulièrement pénibles à comparer.
>
> Aux erreurs de représentation s'ajoutent les erreurs de conversion. La représentation interne des nombres flottants ne permet parfois même pas de stocker un nombre qui, dans notre écriture décimale, tombe juste (par exemple 1.2 ne tombe pas juste en représentation binaire, et `1.2 * 10 - 12` ne fait pas 0...). De même, les conversions de et vers les représentations textuelles des nombres flottants induisent très souvent des erreurs de conversion.
>
> De ce fait, on n'utilisera les nombres flottants qu'à bon escient. Ils ne faut *jamais* utiliser de type en virgule flottante pour stocker des données prenant une plage de valeurs discrètes ou dans les programmes nécessitant une grande rigueur numérique. Ainsi, on ne stockera *jamais* un montant monétaire dans un flottant (bien que de nombreux programmeurs l'ont fait et continueront sans doute à le faire... et à souffrir pour des problèmes qu'ils pouvaient éviter). On ne stockera non

plus *jamais* une durée informatique ou un temps dans un nombre flottant (bien que nombre de programmes sous Windows le fassent par exemple).

À ces types fondamentaux s'ajoutent des types dérivés à l'aide des mots clés « `long` » et « `short` ». Ces types dérivés se distinguent des types de base par l'étendue des plages de valeurs qu'ils peuvent stocker. Par exemple, le type long int est plus grand que le type int. Inversement, le type short int est plus court que le type int. De même, le type long double est plus grand que le type double.

Il est également possible de n'utiliser que des plages de valeurs positives pour les types intégraux (sauf pour les booléens). Pour cela, on peut utiliser les mots clés `signed` et `unsigned`. Les types non signés ne peuvent contenir de valeurs négatives, mais leur valeur maximale est jusqu'à deux fois plus grande que pour leur homologues signés. Nous verrons plus en détail les limites et les plages de valeurs des types de données dans le Chapitre 3.

Exemple 1-5. Types signés et non signés

```
unsigned char
signed char
unsigned wchar_t
signed wchar_t
unsigned short int
signed short int
unsigned int
signed int
unsigned long int
long unsigned int
```

Lorsque le type auquel ils s'appliquent est le type int, les mots clés `signed`, `unsigned`, `short` et `long` peuvent être utilisés seuls, sans le nom de type. Ainsi, le type short int peut être noté simplement short.

Note : Le type int est signé, il est donc inutile de le préciser. En revanche, les types char et wchar_t n'ont pas de signe à proprement parler en C++, et leur signe est indéterminé en C (c'est-à-dire qu'il dépend du compilateur ou des options de compilation). Par conséquent, on précisera toujours le signe lorsque l'on désirera utiliser un type de caractères signé.

Il n'y a pas de type court pour les types réels. De plus, le type long du type float est le type double. Le mot clé `short` n'est donc pas utilisable avec les types réels, et le mot clé `long` ne l'est pas avec le type float.

Il n'y a pas non plus de type de base permettant de manipuler les chaînes de caractères. En C/C++, les chaînes de caractères sont en réalité des tableaux de caractères. Vous trouverez plus loin pour de plus amples informations sur les chaînes de caractères et les tableaux.

1.4.3. Notation des valeurs

La manière de noter les valeurs numériques dépend de leur type.

1.4.3.1. Notation des valeurs booléennes

Les valeurs booléennes se notent de manière extrêmement simple, puisque seules deux valeurs sont autorisées. La valeur fausse se note « `false` » et la valeur vraie « `true` » :

```
bool vrai = true;
bool faux = false;
```

1.4.3.2. Notation des valeurs entières

Pour les entiers, il est possible de spécifier les valeurs numériques en base dix, en base seize ou en base huit. L'écriture des entiers se fait, en fonction de la base choisie, de la manière suivante :

- Avec les chiffres de '`0`' à '`9`' et les signes '`+`' (facultatif) et '`-`' pour la base 10 (notation décimale).

  Exemple 1-6. Notation des entiers en base 10

  ```
  int i = 12354;
  int j = -2564;
  ```

- Avec les chiffres '`0`' à '`9`' et '`A`' à '`F`' ou '`a`' à '`f`' pour la base 16 (avec les conventions A=a=10, B=b=11, ... F=f=15). Les entiers notés en hexadécimal devront toujours être précédés de « `0x` ». Les nombres hexadécimaux ne sont pas signés.

  Exemple 1-7. Notation des entiers en base 16

  ```
  int i_hexa = 0x1AE;
  ```

- Avec les chiffres de '`0`' à '`7`' pour la base 8 (notation octale). Les nombres octaux doivent être précédés d'un 0, et ne sont pas non plus signés.

  Exemple 1-8. Notation des entiers en base 8

  ```
  int i_oct1 = 01;
  int i_oct2 = 0154;
  ```

1.4.3.3. Notation des valeurs en virgule flottantes

Les nombres à virgule flottante (pseudo réels) se notent de la manière suivante :

```
[signe] chiffres [.[chiffres]][e|E [signe] exposant][f|l]
```

où `signe` indique le signe. On emploie les signes '`+`' (facultatif) et '`-`' aussi bien pour la mantisse que pour l'exposant. '`e`' ou '`E`' permet de donner l'exposant du nombre flottant. L'exposant est facultatif. Si on ne donne pas d'exposant, on doit donner des chiffres derrière la virgule avec un point et ces chiffres. Le suffixe '`f`' permet de préciser si le nombre est de type float, et le suffixe '`l`' permet de

préciser si le nombre est de type long double. En l'absence de qualificatif, les valeurs flottantes sont de type double.

Les chiffres après la virgule sont facultatifs, mais pas le point. Si on ne met ni le point, ni la mantisse, le nombre est un entier décimal.

Exemple 1-9. Notation des flottants

```
float f = -123.56f;
double d = 12e-12;
double i = 2.;
```

« 2 » est entier, « 2. » est flottant.

1.4.3.4. Notation des caractères

Les caractères se notent entre guillemets simples :

```
char c1 = 'A';
char c2 = 'c';
char c3 = '(';
```

On peut obtenir un caractère non accessible au clavier en donnant son code en octal, précédé du caractère '\'. Par exemple, le caractère 'A' peut aussi être noté '\101'. Remarquez que cette notation est semblable à la notation des nombres entiers en octal, et que le '0' initial est simplement remplacé par un '\'. Il est aussi possible de noter les caractères avec leur code en hexadécimal, à l'aide de la notation « \xNN », où NN est le code hexadécimal du caractère.

Il existe également des séquences d'échappement particulières qui permettent de coder certains caractères spéciaux plus facilement. Les principales séquences d'échappement sont les suivantes :

Séquence	Signification
'\a'	Bip sonore
'\b'	Retour arrière
'\f'	Début de page suivante
'\n'	Saut de ligne (sans retour de chariot)
'\r'	Retour à la ligne (sans saut de ligne)
'\t'	Tabulation horizontale
'\w'	Tabulation verticale

D'autres séquences d'échappement sont disponibles, afin de pouvoir représenter les caractères ayant une signification particulière en C :

Séquence	Signification
'\\'	Le caractère '\'
'\"'	Le caractère '"'
'\''	Le caractère '''

Enfin, les valeurs des caractères larges de type wchar_t sont notées de la même manière que les valeurs des caractères simples, mais doivent être précédées de la lettre 'L'. Par exemple :

```
wchar_t c1 = L'A';
wchar_t c2 = L'c';
wchar_t c3 = L'(';
```

1.4.3.5. Notation des chaînes de caractères

En C/C++, les chaînes de caractères apparaissent comme des séquences de caractères consécutifs, terminées par un caractère nul. Ces chaînes peuvent être écrites en spécifiant l'ensemble des caractères de la chaîne entre doubles guillemets :

```
"Exemple de chaîne de caractères..."
```

Vous remarquerez que le caractère nul terminal n'est pas spécifié. Le compilateur l'ajoute automatiquement, il ne faut donc pas le faire.

Les caractères spéciaux peuvent être utilisés directement dans les chaînes de caractères constantes :

```
"Ceci est un saut de ligne :\nCeci est à la ligne suivante."
```

> **Note :** Attention : du fait que le caractère nul est utilisé en tant que marqueur de fin de chaîne, il ne faut pas l'utiliser dans une chaîne de caractères. Même si le compilateur prendra en compte les caractères qui suivent ce caractère nul, les fonctions de la bibliothèque C ne parviendront pas à manipuler correctement la chaîne et ne traiteront que les caractères précédant le caractère nul.

Si une chaîne de caractères constante est trop longue pour tenir sur une seule ligne, on peut concaténer plusieurs chaînes en les juxtaposant :

```
"Ceci est la première chaîne "
    "ceci est la deuxième."
```

produit la chaîne de caractères complète suivante :

```
"Ceci est la première chaîne ceci est la deuxième."
```

Vous noterez que dans ce cas le compilateur n'insère pas de caractère nul entre les deux chaînes.

Enfin, les chaînes de caractères de type wchar_t doivent être précédées du préfixe 'L' :

```
L"Ceci est une chaîne de wchar_t."
```

1.5. Les instructions

Les instructions sont les éléments de base du programme. Tout les traitements des programmes C/C++ sont donc effectués par des suites d'instructions.

Il existe plusieurs types d'instructions en C/C++ :

- les instructions simples ;
- les instructions composées ;
- les structures de contrôle.

1.5.1. Les instructions simples

Les instructions simples sont identifiées par le point virgule. C'est ce caractère qui marque la fin d'une instruction. Le programme evalue l'expression délimitée par l'instruction précédente et ce point virgule. Cette expression peut être vide, le résultat de l'invocation d'une fonction, ou une combinaison d'expressions plus simples par des opérateurs.

Exemple 1-10. Exemple d'instructions

```
;                           /* Instruction réduite à sa plus simple expression
                               (ne fait rien) */
printf("Hello World!\n");   /* Instruction appelant une fonction */
i = j + m;                  /* Instruction évaluant une combinaison d'opérateurs */
```

Les principales opérations que l'on peut réaliser dans une expression sont les suivantes :

Tableau 1-2. Opérateurs du langage C/C++

Opérateur	Signification
`a = b`	Opérateur d'affectation. `a` reçoit la valeur de `b`. Renvoie la valeur de `b`.
`a + b`	Opérateur d'addition. Renvoie la valeur de la somme de `a` et de `b`.
`a - b`	Opérateur de soustraction. Renvoie la valeur de la soustraction de `a` et de `b`.
`a * b`	Opérateur de multiplication. Renvoie la valeur du produit de `a` et de `b`.
`a / b`	Opérateur de division. Pour les entiers, renvoie la valeur de la division euclidienne (division entière) de `a` et de `b`. Pour les nombres à virgule flottante, renvoie la valeur de la division de `a` et de `b`.
`a % b`	Opérateur de reste de division. Cet opérateur renvoie la valeur du reste de la division euclidienne des entiers `a` et `b`.
`a & b` ou `a bitand b`	Opérateur de conjonction. Renvoie la valeur du « et » logique bit à bit de `a` et `b`. Chaque bit du résultat est calculé à partir des bits correspondants de `a` et de `b` avec la règle suivante : 1 et 1 = 1, (0 et x) = (x et 0) = 0.
`a | b` ou `a bitor b`	Opérateur de disjonction. Renvoie la valeur du « ou » logique bit à bit de `a` et de `b`. Chaque bit du résultat est calculé à partir des bits correspondants de `a` et de `b` avec la règle suivante : 0 ou 0 = 0, (1 ou x) = (x ou 1) = 1.

Opérateur	Signification
`a ^ b` ou `a xor b`	Opérateur de disjonction exclusive. Renvoie la valeur du « ou exclusif » logique bit à bit (généralement appelé « xor ») de de `a` et de `b`. Chaque bit du résultat est calculé à partir des bits correspondants de `a` et de `b` avec la règle suivante : 1 xor 1 = 0, 0 xor 0 = 0 et (1 xor 0) = (0 xor 1) = 1.
`~a` ou `compl a`	Opérateur de négation. Renvoie la valeur dont chaque bit est l'inverse du bit correspondant de `a` (0 devient 1 et vice versa).
`a << b`	Opérateur de décallage binaire à gauche. Renvoie la valeur de `a` dont les bits ont été décallés vers les bits de poids fort `b` fois. Les bits de poids faible insérés sont nuls. Comme l'ajout d'un 0 à la droite d'un nombre revient à le multiplier par sa base, cet opérateur permet de multiplier `a` par deux à la puissance `b` (s'il n'y a pas de débordement des bits par la gauche bien entendu).
`a >> b`	Opérateur de décallage binaire à droite. Renvoie la valeur de `a` dont les bits ont été décallés vers les bits de poids faible `b` fois. Les bits de poids fort insérés sont nuls. Cela revient à diviser `a` par deux à la puissance `b`.
`++a`	Opérateur de pré-incrémentation. Ajoute 1 à `a` et retourne le résultat obtenu.
`a++`	Opérateur de post-incrémentation. Ajoute 1 à `a` et renvoie la valeur précédente de celui-ci.
`--a`	Opérateur de pré-décrémentation. Ôte 1 de `a` et renvoie la valeur obtenue.
`a--`	Opérateur de post-décrémentation. Öte 1 de `a` et renvoie la valeur précédente de celui-ci.
`a ? b : c`	Opérateur ternaire. Évalue la valeur de l'expression `a` et, selon qu'elle est vraie ou non, évalue l'expression `c` ou l'expression `b`. Cet opérateur permet donc de faire un test et de choisir une valeur ou une autre selon le résultat de ce test.
`a , b`	Opérateur virgule. Évalue les expressions `a` et `b` et retourne la valeur de `b`.

On notera que les affectations ne sont pas des instructions. Ce sont bien des opérations, dont la valeur est la valeur affectée. Il est donc possible de réutiliser cette valeur dans une expression plus complexe. Un cas intéressant est l'utilisation de cette valeur pour faire une autre affectation : il est ainsi possible d'effectuer des affectations multiples :

```
i=j=k=m=0;   /* Annule les variables i, j, k et m. */
```

De tous les opérateurs, c'est sans doute l'opérateur d'affectation qui est le plus utilisé. C'est pour cette raison que le C et le C++ proposent des opérateurs d'*affectations composées*. Une affectation composée est une opération permettant de réaliser en une seule étape une opération normale et l'affectation de son résultat dans la variable servant de premier opérande. Les affectations composées utilisent la syntaxe suivante :

```
variable op_aff valeur
```

où `op_aff` est l'un des opérateurs suivants : '`+=`', '`-=`', '`*=`', etc. Cette syntaxe est strictement équivalente à :

```
variable = variable op valeur
```

et permet donc de modifier la valeur de `variable` en lui appliquant l'opérateur `op`.

Exemple 1-11. Affectation composée

```
i*=2;         /* Multiplie i par 2 : i = i * 2. */
```

> **Note :** Les opérateurs '`&=`', '`|=`' et '`^=`' peuvent également s'écrire respectivement '`and_eq`', '`or_eq`' et '`xor_eq`'.

Les opérateurs d'incrémentation et de décrémentation `++` et `--` peuvent s'appliquer comme des préfixes ou des suffixes sur les variables. Lorsqu'ils sont en préfixe, la variable est incrémentée ou décrémentée, puis sa valeur est renvoyée. S'ils sont en suffixe, la valeur de la variable est renvoyée, puis la variable est incrémentée ou décrémentée. Par exemple :

```
int i=2,j,k;

j=++i;   /* À la fin de cette instruction, i et j valent 3. */
k=j++;   /* À la fin de cette ligne, k vaut 3 et j vaut 4. */
```

> **Note :** On prendra garde à n'utiliser les opérateurs d'incrémentation et de décrémentation postfixés que lorsque cela est réellement nécessaire. En effet, ces opérateurs doivent contruire un objet temporaire pour renvoyer la valeur de la variable avant incrémentation ou décrémentation. Si cet objet temporaire n'est pas utilisé, il est préférable d'utiliser les versions préfixées de ces opérateurs.

L'opérateur ternaire d'évaluation conditionnelle `?:` est le seul opérateur qui demande 3 paramètres (à part l'opérateur fonctionnel `()` des fonctions, qui admet n paramètres, et que l'on décrira plus tard). Cet opérateur permet de réaliser un test sur une condition et de calculer une expression ou une autre selon le résultat de ce test. La syntaxe de cet opérateur est la suivante :

```
test ? expression1 : expression2
```

Dans cette syntaxe, `test` est évalué en premier. Son résultat doit être booléen ou entier. Si `test` est vrai (ou si sa valeur est non nulle), `expression1` est calculée et sa valeur est renvoyée. Sinon, c'est la valeur de `expression2` qui est renvoyée. Par exemple, l'expression :

```
Min=(i<j)?i:j;
```

calcule le minimum de `i` et de `j`.

L'opérateur virgule, quant à lui, permet d'évaluer plusieurs expressions successivement et de renvoyer la valeur de la dernière expression. La syntaxe de cet opérateur est la suivante :

```
expression1,expression2[,expression3[...]]
```

où `expression1`, `expression2`, etc. sont les expressions à évaluer. Les expressions sont évaluées de gauche à droite, puis le type et la valeur de la dernière expression sont utilisés pour renvoyer le résultat. Par exemple, à l'issue des deux lignes suivantes :

```
double r = 5;
```

```
int i = r*3,1;
```

`r` vaut 5 et `i` vaut 1. `r*3` est calculé pour rien.

> **Note :** Ces deux derniers opérateurs peuvent nuire gravement à la lisibilité des programmes. Il est toujours possible de réécrire les lignes utilisant l'opérateur ternaire avec un test (voir le Chapitre 2 pour la syntaxe des tests en C/C++). De même, on peut toujours décomposer une expression utilisant l'opérateur virgule en deux instructions distinctes. Ce dernier opérateur ne devra donc jamais être utilisé.

1.5.2. Les instructions composée

Il est possible de créer des *instructions composées*, constituées d'instructions plus simples. Les instructions composées se présentent sous la forme de blocs d'instructions où les instructions contenues sont encadrées d'accolades ouvrantes et fermantes (caractères '`{` et '`}`').

Exemple 1-12. Instruction composée

```
{
    i=1;
    j=i+3*g;
}
```

> **Note :** Un bloc d'instructions est considéré comme une instruction unique. Il est donc inutile de mettre un point virgule pour marquer l'instruction, puisque le bloc lui-même est une instruction.

1.5.3. Les structures de contrôle

Enfin, il existe tout un jeu d'instructions qui permettent de modifier le cours de l'exécution du programme, comme les tests, les boucles et les sauts. Ces instructions seront décrites en détail dans le chapitre traitant des structures de contrôle.

1.6. Les fonctions et les procédures

Les *fonctions* sont des groupements d'instructions qui peuvent prendre des paramètres en entrée et qui retournent une valeur. Elles peuvent être appelées plusieurs fois, avec des valeurs de paramètres différentes. Elles permettent donc d'implémenter un algorithme de calcul et de le réutiliser pour différents jeux de paramètres.

Les *procédures* sont des groupements d'instructions qui effectuent une tâche, mais qui n'ont a priori pas pour but de calculer quelque chose. Elles ne retournent donc pas de valeur. Toutefois, elles peuvent prendre des arguments, dont les valeurs permettent d'en modifier le comportement.

Comme on l'a vu, le programme principal est lui-même constitué d'une fonction spéciale, la fonction `main`. Cette fonction peut appeler d'autres fonctions ou procédures, qui elles-mêmes peuvent appeler

encore d'autres fonctions. L'exécution du programme constitue donc une suite d'appels de fonctions et de procédures.

1.6.1. Définition des fonctions et des procédures

La définition des fonctions se fait comme suit :

```
type identificateur(paramètres)
{
    ...   /* Instructions de la fonction. */
}
```

type est le type de la valeur renvoyée, `identificateur` est le nom de la fonction, et `paramètres` est une liste de paramètres. Les règles de nommage des identificateurs de fonctions sont les mêmes que pour les identificateurs de variables (voir Section 1.4).

La syntaxe de la liste de paramètres est la suivante :

```
type variable [= valeur] [, type variable [= valeur] [...]]
```

où type est le type du paramètre *`variable`* qui le suit et `valeur` sa valeur par défaut. La valeur par défaut d'un paramètre est la valeur que ce paramètre prend si aucune valeur ne lui est attribuée lors de l'appel de la fonction.

> **Note :** L'initialisation des paramètres de fonctions n'est possible qu'en C++, le C n'accepte pas cette syntaxe.

La valeur de la fonction à renvoyer est spécifiée en utilisant le mot clé `return`, dont la syntaxe est :

```
return valeur;
```

Exemple 1-13. Définition de fonction

```
int somme(int i, int j)
{
    return i+j;
}
```

En C/C++, les procédures sont réalisées simplement en définissant une fonction qui ne retourne pas de valeur. Pour cela, on utilise le type de retour void. Dans ce cas, il n'est pas nécessaire de mettre une instruction `return` en fin de procédure. Cela est cependant faisable, il suffit de ne pas donner de valeur dans l'instruction `return`.

Si une fonction ou une procédure ne prend pas de paramètres d'entrée, sa liste de paramètres peut être omise. Il est également possible de spécifier explicitement que la fonction ne prend pas de paramètres en utilisant le mot clé void. .

Exemple 1-14. Définition de procédure

```
void rien()     /* Fonction n'attendant pas de paramètres */
{               /* et ne renvoyant pas de valeur. */
    return;     /* Cette ligne est facultative. */
```

```
}
```

Note : Il est spécifié dans la norme du C++ que la fonction `main` ne doit pas renvoyer le type void. En pratique cependant, beaucoup de compilateurs l'acceptent également.

1.6.2. Appel des fonctions et des procédures

L'appel d'une fonction ou d'une procédure se fait en donnant son nom, puis les valeurs de ses paramètres entre parenthèses. Attention ! S'il n'y a pas de paramètres, il faut quand même mettre les parenthèses, sinon la fonction n'est pas appelée.

Exemple 1-15. Appel de fonction

```
int i=somme(2,3);
rien();
```

Comme vous pouvez le constater, une fonction peut être utilisée en lieu et place de sa valeur de retour dans une expression. Dans l'exemple précédent, le résultat de la fonction `somme` est affecté à la variable `i`. En revanche, les procédures doivent être utilisées dans des instructions simples, puisqu'elles ne retournent aucune valeur.

En C++ (et uniquement en C++), les paramètres qui ont des valeurs par défaut dans la déclaration de la fonction ou de la procédure peuvent être omis lors de l'appel. Les valeurs que ces paramètres auront dans la fonction seront alors les valeurs par défaut indiquées dans la déclaration. Dès qu'un paramètre est manquant lors de l'appel, tous les paramètres qui le suivent doivent eux aussi être omis et prendre leur valeur par défaut. Autrement dit, le C++ ne permet pas d'effectuer des appels en spécifiant les valeurs des paramètres explicitement par leurs noms, seule la position des paramètres d'appel indique de quel paramètre dans la définition de la fonction il s'agit. Il en résulte que seuls les derniers paramètres d'une fonction peuvent avoir des valeurs par défaut. Par exemple :

```
int test(int i = 0, int j = 2)
{
    return i/j;
}

int main(void)
{
    int resultat1 = test(8);  /* Appel de test(8, 2) */
    int resultat2 = test();   /* Appel de test(0, 2) */
    return EXIT_SUCCESS;
}
```

L'appel de la fonction `test(8)` est valide. Comme on ne précise pas le dernier paramètre, *`j`* est initialisé à `2`. Le premier résultat est donc `4`. De même, l'appel `test()` est valide : dans ce cas *`i`* vaut `0` et *`j`* vaut `2`. En revanche, il est impossible d'appeler la fonction `test` en ne précisant que la valeur de *`j`*. Enfin, l'expression « `int test(int i=0, int j) {...}` » serait invalide, car si on ne passait pas deux paramètres, *`j`* ne serait pas initialisé.

Il est possible, pour une fonction ou une procédure, de s'appeler soi-même, soit directement, soit indirectement via une autre fonction. De telles fonctions sont appelées des *fonctions récursives*.

Les appels récursifs doivent être limités, car lors de chaque appel, la liste des paramètres fournis est mémorisée. Un appel récursif infini induit donc une consommation mémoire infinie (ce qui se traduit généralement par un débordement de pile au niveau du processus). Il est donc nécessaire, lorsqu'on réalise une fonction ou une procédure récursive, d'avoir une condition de sortie qui sera toujours vérifiée au bout d'un certain nombre d'appel.

L'archétype de la fonction récursive est la fonction factorielle, qui calcule le produit des n premiers nombres entiers (avec pour convention que factorielle de 0 et de 1 vallent 1) :

```
int factorielle(int n)
{
    return (n > 1) ? n * factorielle(n - 1) : 1;
}
```

Note : Nous verrons dans le Chapitre 2 la manière d'écrire un test sans avoir recours à l'opérateur ternaire `?:`. Comme l'écriture précédente le montre, cet opérateur ne facilite que très rarement la lisibilité d'un programme...

1.6.3. Notion de déclaration

Toute fonction ou procédure doit être *déclarée* avant d'être appelée pour la première fois. La *définition* peut faire office de *déclaration*, toutefois il peut se trouver des situations où une fonction ou une procédure doit être appelée dans une autre fonction définie avant elle. Comme cette fonction n'est pas définie au moment de l'appel, elle doit être déclarée.

De même, il est courant d'avoir défini une fonction dans un fichier et de devoir faire l'appel de cette fonction à partir d'un autre fichier. Il est donc là aussi nécessaire de déclarer cette fonction.

Le rôle des déclarations est donc de signaler l'existence des fonctions et des procédures au compilateur afin de pouvoir les utiliser, tout en reportant leurs définitions plus loin ou dans un autre fichier. Cela permet de vérifier que les paramètres fournis à une fonction correspondent bien à ce qu'elle attend, et que la valeur de retour est correctement utilisée après l'appel.

La syntaxe de la déclaration d'une fonction est la suivante :

```
type identificateur(paramètres);
```

où type est le type de la valeur renvoyée par la fonction (éventuellement `void`), `identificateur` est son nom et `paramètres` la liste des types des paramètres que la fonction admet, éventuellement avec leurs valeurs par défaut, et séparés par des virgules.

Exemple 1-16. Déclaration de fonction

```
int Min(int, int);          /* Déclaration de la fonction minimum */
                            /* définie plus loin. */
/* Fonction principale. */
int main(void)
{
    int i = Min(2,3);       /* Appel à la fonction Min, déjà
                               déclarée. */
    return 0;
}
```

```
/* Définition de la fonction min. */
int Min(int i, int j)
{
    return i<j ? i : j;
}
```

Si l'on donne des valeurs par défaut différentes aux paramètres dans plusieurs déclarations, les valeurs par défaut utilisées sont celles de la déclaration visible lors de l'appel. Si plusieurs déclarations sont visibles et entrent en conflit au niveau des valeurs par défaut des paramètres de la fonction ou de la procédure, le compilateur ne saura pas quelle déclaration utiliser et signalera une erreur à la compilation.

Il est possible de compléter la liste des valeurs par défaut de la déclaration d'une fonction ou d'une procédure dans sa définition. Dans ce cas, les valeurs par défaut spécifiées dans la définition ne doivent pas entrer en conflit avec celles spécifiées dans la déclaration visible au moment de la définition, faute de quoi le compilateur signalera une erreur.

1.6.4. Surcharge des fonctions

Il est interdit en C de définir plusieurs fonctions qui portent le même nom. En C++, cette interdiction est levée, moyennant quelques précautions. Le compilateur peut différencier deux fonctions en regardant le type des paramètres qu'elle reçoit. La liste de ces types s'appelle la *signature* de la fonction. En revanche, le type du résultat de la fonction ne permet pas de l'identifier, car le résultat peut ne pas être utilisé ou peut être converti en une valeur d'un autre type avant d'être utilisé après l'appel de cette fonction.

Il est donc possible de faire des fonctions de même nom (on les appelle alors des *surcharges*) si et seulement si toutes les fonctions portant ce nom peuvent être distinguées par leurs signatures. La surcharge qui sera appelée sera celle dont la signature est la plus proche des valeurs passées en paramètre lors de l'appel.

Exemple 1-17. Surcharge de fonctions

```
float test(int i, int j)
{
    return (float) i+j;
}

float test(float i, float j)
{
    return i*j;
}
```

Ces deux fonctions portent le même nom, et le compilateur les acceptera toutes les deux. Lors de l'appel de `test(2,3)`, ce sera la première qui sera appelée, car `2` et `3` sont des entiers. Lors de l'appel de `test(2.5,3.2)`, ce sera la deuxième, parce que `2.5` et `3.2` sont réels. Attention ! Dans un appel tel que `test(2.5,3)`, le flottant `2.5` sera converti en entier et la première fonction sera appelée. Il convient donc de faire très attention aux mécanismes de surcharge du langage, et de vérifier les règles de priorité utilisées par le compilateur.

On veillera à ne pas utiliser des fonctions surchargées dont les paramètres ont des valeurs par défaut, car le compilateur ne pourrait pas faire la distinction entre ces fonctions. D'une manière générale, le compilateur dispose d'un ensemble de règles (dont la présentation dépasse le cadre de ce livre) qui

lui permettent de déterminer la meilleure fonction à appeler étant donné un jeu de paramètres. Si, lors de la recherche de la fonction à utiliser, le compilateur trouve des ambiguïtés, il génére une erreur.

Le C++ considère les types char et wchar_t comme des types à part entière, utilisés pour stocker des caractères. Ils n'ont donc pas de signe en soi, et le compilateur considère donc comme des types distincts les versions signées et non signées du type de base. Cela signifie que le compilateur traitent les types char, unsigned char et signed char comme des types différents, et il en est de même pour les types wchar_t, signed wchar_t et unsigned wchar_t. Cette distinction est importante dans la détermination de la signature des fonctions, puisqu'elle permet de ce fait de faire des surcharges de fonctions pour ces types de données.

1.6.5. Fonctions inline

Le C++ dispose du mot clef `inline`, qui permet de modifier la méthode d'implémentation des fonctions. Placé devant la déclaration d'une fonction, il propose au compilateur de ne pas instancier cette fonction. Cela signifie que l'on désire que le compilateur remplace l'appel de la fonction par le code correspondant. Si la fonction est grosse ou si elle est appelée souvent, le programme devient plus gros, puisque la fonction est réécrite à chaque fois qu'elle est appelée. En revanche, il devient nettement plus rapide, puisque les mécanismes d'appel de fonctions, de passage des paramètres et de récupération de la valeur de retour sont ainsi évités. De plus, le compilateur peut effectuer des optimisations additionnelles qu'il n'aurait pas pu faire si la fonction n'était pas inlinée. En pratique, on réservera cette technique pour les petites fonctions appelées dans du code devant être rapide (à l'intérieur des boucles par exemple), ou pour les fonctions permettant de lire des valeurs dans des variables.

Cependant, il faut se méfier. Le mot clé `inline` est une autorisation donnée au compilateur pour faire des fonctions `inline`. Il n'y est pas obligé. La fonction peut donc très bien être implémentée classiquement. Pire, elle peut être implémentée des deux manières, selon les mécanismes d'optimisation du compilateur. De même, le compilateur peut également inliner automatiquement et de manière transparente les fonctions normales afin d'optimiser les performances du programme.

De plus, il faut connaître les restrictions des fonctions `inline` :

- elles ne peuvent pas être récursives ;
- elles ne sont pas instanciées, donc on ne peut pas faire de pointeur sur une fonction `inline`.

Si l'une de ces deux conditions n'est pas vérifiée pour une fonction, le compilateur l'implémentera classiquement (elle ne sera donc pas `inline`).

Enfin, du fait que les fonctions `inline` sont insérées telles quelles aux endroits où elles sont appelées, il est nécessaire qu'elles soient complètement définies avant leur appel. Cela signifie que, contrairement aux fonctions classiques, il n'est pas possible de se contenter de les déclarer pour les appeler, et de fournir leur définition dans un fichier séparé. Dans ce cas en effet, le compilateur générerait des références externes sur ces fonctions, et n'insérerait pas leur code. Il peut, s'il voit ensuite la définition de la fonction, l'instancier sans l'inliner, mais dans le cas contraire, le code de la fonction ne sera pas compilé. Les références à la fonction inline ne seront donc pas résolues à l'édition de liens, et le programme ne pourra pas être généré. Les notions de compilation dans des fichiers séparés et d'édition de liens seront présentées en détail dans le Chapitre 6.

Exemple 1-18. Fonction inline

```
inline int Max(int i, int j)
{
```

```
    return i>j ? i : j;
}
```

Pour ce type de fonction, il est tout à fait justifié d'utiliser le mot clé `inline`.

1.6.6. Fonctions statiques

Par défaut, lorsqu'une fonction est définie dans un fichier C/C++, elle peut être utilisée dans tout autre fichier pourvu qu'elle soit déclarée avant son utilisation. Dans ce cas, la fonction est dite *externe*. Il peut cependant être intéressant de définir des fonctions locales à un fichier, soit afin de résoudre des conflits de noms (entre deux fonctions de même nom et de même signature mais dans deux fichiers différents), soit parce que la fonction est uniquement d'intérêt local. Le C et le C++ fournissent donc le mot clé `static` qui, une fois placé devant la définition et les éventuelles déclarations d'une fonction, la rend unique et utilisable uniquement dans ce fichier. À part ce détail, les fonctions statiques s'utilisent exactement comme des fonctions classiques.

Exemple 1-19. Fonction statique

```
/* Déclaration de fonction statique : */
static int locale1(void);

/* Définition de fonction statique : */
static int locale2(int i, float j)
{
    return i*i+j;
}
```

Les techniques permettant de découper un programme en plusieurs fichiers sources et de générer les fichiers binaires à partir de ces fichiers seront décrites dans le chapitre traitant de la modularité des programmes.

> **Note :** Le C++ dispose d'un mécanisme plus souple d'isolation des entités propres à un fichier, via le mécanisme des espaces de nommages. De plus, le mot clef `static` a une autre signification en C++ dans un autre contexte d'utilisation. L'emploi de ce mot clé pour rendre local à un fichier une fonction est donc déconseillée en C++, et l'on utilisera de préférence les mécanismes présentés dans le Chapitre 10.

1.6.7. Fonctions prenant un nombre variable de paramètres

En général, les fonctions ont un nombre constant de paramètres. Pour les fonctions qui ont des paramètres par défaut en C++, le nombre de paramètres peut apparaître variable à l'appel de la fonction, mais en réalité, la fonction utilise toujours le même nombre de paramètres.

Le C et le C++ disposent toutefois d'un mécanisme qui permet au programmeur de réaliser des fonctions dont le nombre et le type des paramètres sont variables. Nous verrons plus loin que les fonctions d'entrée / sortie du C sont des fonctions dont la liste des arguments n'est pas fixée, cela afin de pouvoir réaliser un nombre arbitraire d'entrées / sorties, et ce sur n'importe quel type prédéfini.

En général, les fonctions dont la liste des paramètres est arbitrairement longue disposent d'un critère pour savoir quel est le dernier paramètre. Ce critère peut être le nombre de paramètres, qui peut être fourni en premier paramètre à la fonction, ou une valeur de paramètre particulière qui détermine la fin

de la liste par exemple. On peut aussi définir les paramètres qui suivent le premier paramètre à l'aide d'une chaîne de caractères.

Pour indiquer au compilateur qu'une fonction peut accepter une liste de paramètres variable, il faut simplement utiliser des points de suspensions dans la liste des paramètres :

```
type identificateur(paramètres, ...)
```

dans les déclarations et la définition de la fonction. Dans tous les cas, il est nécessaire que la fonction ait au moins un paramètre classique. Les paramètres classiques doivent impérativement être avant les points de suspensions.

La difficulté apparaît en fait dans la manière de récupérer les paramètres de la liste de paramètres dans la définition de la fonction. Les mécanismes de passage des paramètres étant très dépendants de la machine (et du compilateur), un jeu de macros a été défini dans le fichier d'en-tête `stdarg.h` pour faciliter l'accès aux paramètres de la liste. Pour en savoir plus sur les macros, consulter le Chapitre 5. Pour l'instant, sachez seulement que l'inclusion du fichier d'en-tête `stdarg.h` vous permettra d'utiliser le type va_list et les expressions `va_start`, `va_arg` et `va_end` pour récupérer les arguments de la liste de paramètres variable, un à un.

Le principe est simple. Dans la fonction, vous devez déclarer une variable de type va_list. Puis, vous devez initialiser cette variable avec la syntaxe suivante :

```
va_start(variable, paramètre);
```

où `variable` est le nom de la variable de type va_list que vous venez de créer, et `paramètre` est le dernier paramètre classique de la fonction. Dès que `variable` est initialisée, vous pouvez récupérer un à un les paramètres à l'aide de l'expression suivante :

```
va_arg(variable, type)
```

qui renvoie le paramètre en cours avec le type type et met à jour `variable` pour passer au paramètre suivant. Vous pouvez utiliser cette expression autant de fois que vous le désirez, elle retourne à chaque fois un nouveau paramètre. Lorsque le nombre de paramètres correct a été récupéré, vous devez détruire la variable `variable` à l'aide de la syntaxe suivante :

```
va_end(variable);
```

Il est possible de recommencer ces étapes autant de fois que l'on veut, la seule chose qui compte est de bien faire l'initialisation avec `va_start` et de bien terminer la procédure avec `va_end` à chaque fois.

Exemple 1-20. Fonction à nombre de paramètres variable

```
#include <stdarg.h>

/* Fonction effectuant la somme de "compte" paramètres : */
double somme(int compte, ...)
{
    double resultat=0;      /* Variable stockant la somme. */
    va_list varg;           /* Variable identifiant le prochain
                               paramètre. */
    va_start(varg, compte); /* Initialisation de la liste. */
    while (compte!=0)       /* Parcours de la liste. */
    {
        resultat=resultat+va_arg(varg, double);
```

```
        compte=compte-1;
    }
    va_end(varg);           /* Terminaison. */
    return resultat;
}
```

La fonction `somme` effectue la somme de compte flottants (float ou double) et la renvoie dans un double. Pour plus de détails sur la structure de contrôle `while`, voir Section 2.2.2.

> **Note :** Il existe une restriction sur les types des paramètres des listes variables d'arguments. En effet, seuls quelques types de données sont utilisables, les autres étant convertis automatiquement vers le premier type de données autorisé capable de stocker l'ensemble des valeurs du paramètre. Ces opérations sont décrites en détail dans la Section 3.5.

1.7. Les entrées / sorties en C

Nous avons présenté au début de ce chapitre la fonction `printf` dont le rôle est de permettre d'écrire sur le flux de sortie standard des données formatées. Nous avions également indiqué que nombre de programmes récupèrent les données sur lesquelles ils doivent travailler sur le flux d'entrée standard, et envoient les résultats sur le flux de sortie standard.

Nous allons donc voire à présent un peu plus en détail les notions de flux d'entrée / sortie standards et décrire de manière plus approfondie les fonctions de la bibliothèque C qui permettent de réaliser ces entrées / sorties.

1.7.1. Généralités sur les flux d'entrée / sortie

Un *flux* est une notion informatique qui permet de représenter un flot de données séquentielles en provenance d'une source de données ou à destination d'une autre partie du système. Les flux sont utilisés pour uniformiser la manière dont les programmes travaillent avec les données, et donc pour simplifier leur programmation. Les fichiers constituent un bon exemple de flux, mais ce n'est pas le seul type de flux existant : on peut traiter un flux de données provenant d'un réseau, d'un tampon mémoire ou de toute autre source de données ou partie du système permettant de traiter les données séquentiellement.

Sur quasiment tous les systèmes d'exploitation, les programmes disposent dès leur lancement de trois flux d'entrée / sortie standards. Généralement, le flux d'entrée standard est associé au flux de données provenant d'un terminal, et le flux de sortie standard à la console de ce terminal. Ainsi, les données que l'utilisateur saisit au clavier peuvent être lues par les programmes sur leur flux d'entrée standard, et ils peuvent afficher leurs résultats à l'écran en écrivant simplement sur leur flux de sortie standard. Le troisième flux standard est le flux d'erreur standard qui, par défaut, est également associé à l'écran, et sur lequel le programme peut écrire tous les messages d'erreur qu'il désire.

La plupart des systèmes permettent de rediriger les flux standards des programmes afin de les faire travailler sur des données provenant d'une autre source de données que le clavier, ou, par exemple, de leur faire enregistrer leurs résultats dans un fichier. Il est même courant de réaliser des « pipelines » de programmes (« *tubes* » en français), où les résultats de l'un sont envoyés dans le flux d'entrée standard de l'autre, et ainsi de suite. Les programmes qui participent à un pipeline sont classiquement appelés des *filtres*, car les plus simples d'entre eux permettent de filtrer les entrées, et éventuellement de les modifier, avant des les renvoyer sur le flux de sortie standard.

> **Note :** La manière de réaliser les redirections des flux standards dépend des systèmes d'exploitation et de leurs interfaces utilisateurs. De plus, les programmes doivent être capables de travailler avec leurs flux d'entrée / sortie standards de manière générique, que ceux-ci soient redirigés ou non. Les techniques de redirection ne seront donc pas décrites plus en détail ici.

Vous remarquerez l'intérêt d'avoir deux flux distincts pour les résultats des programmes et leurs messages d'erreur. Si, lors d'une utilisation normale, ces deux flux se mélangent à l'écran, ce n'est pas le cas lorsque l'on redirige le flux de sortie standard. Seul le flux d'erreur standard est affiché à l'écran dans ce cas, et les messages d'erreur ne se mélangent donc pas aux résultats du programme.

On pourrait penser que les programmes graphiques ne disposent pas de flux d'entrée / sortie standards. Pourtant, c'est généralement le cas. Même si les événements traités par les programmes graphiques dans leur boucle de messages ne proviennent pas du flux d'entrée standard, mais d'une autre source de données spécifique à chaque système, ils peuvent malgré tout utiliser les flux d'entrée / sortie standards si cela s'avère nécessaire. Généralement, les programmes graphiques utilisent le flux de sortie standard pour écrire des messages de fonctionnement ou des messages d'erreur (cette pratique n'est cependant pas courante sous Windows, pour diverses raisons).

1.7.2. Les fonctions d'entrée / sortie de la bibliothèque C

Afin de permettre aux programmes d'écrire sur leurs flux d'entrée / sortie standards, la bibliothèque C définit plusieurs fonctions extrêmement utiles. Les deux principales fonctions sont sans doute les fonctions `printf` et `scanf`. La fonction `printf` (« print formatted » en anglais) permet d'envoyer des données formatées sur le flux de sortie standard, et `scanf` (« scan formatted ») permet de les lire à partir du flux d'entrée standard. Ces fonctions sont déclarées dans le fichier d'en-tête `stdio.h`.

En réalité, ces fonctions ne font rien d'autre que d'appeler deux autres fonctions permettant d'écrire et de lire des données sur un flux quelconque : les fonctions `fprintf` et `fscanf`. Ces fonctions s'utilisent exactement de la même manière que les fonctions `printf` et `scanf`, à ceci près qu'elles prennent en premier paramètre une structure décrivant le flux sur lequel elles travaillent.

Pour les flux d'entrée / sortie standards, la bibliothèque C définit les flux `stdin`, `stdout` et `stderr`, qui correspondent respectivement au flux d'entrée, au flux de sortie et au flux d'erreur standards. Ainsi, tout appel à `scanf` se traduit par un appel à `fscanf` sur le flux `stdin`, et tout appel à `printf` par un appel à `fprintf` sur le flux `stdout`.

> **Note :** Il n'existe pas de fonction permettant d'écrire directement sur le flux d'erreur standard. Par conséquent, pour effectuer de telles écritures, il faut impérativement passer par la fonction `fprintf`, en lui fournissant en paramètre le flux `stderr`.
>
> La description des fonctions de la bibliothèque C standard dépasse de loin le cadre de ce document. Aussi les fonctions de lecture et d'écriture sur les flux ne seront-elles pas décrites plus en détail ici. Seules les fonctions `printf` et `scanf` seront présentées, car elles sont réellement indispensable pour l'écriture d'un programme C et pour la compréhension des exemples des chapitres suivants. Consultez la bibliographie si vous désirez obtenir plus de détails sur la bibliothèque C et sur toutes les fonctions qu'elle contient.
>
> Le C++ dispose également de mécanismes de gestion des flux d'entrée / sortie qui lui sont propres. Ces mécanismes permettent de contrôler plus finement les types des données écrites et lues de et à partir des flux d'entrée / sortie standards. De plus, ils permettent de réaliser les opérations d'écriture et de lecture des données formatées de manière beaucoup plus simple. Cependant, ces mécanismes requièrent des notions objets avancées et ne seront décrits que dans les chapitres dédiés au C++. Comme il est également possible d'utiliser les fonctions `printf` et `scanf` en C++ d'une part, et que, d'autre part, ces fonctions sont essentielles en C, la suite de

cette section s'attachera à leur description. Un chapitre complet est dédié aux mécanismes de gestion des flux du C++ dans la deuxième partie de ce document.

Les fonctions `printf` et `scanf` sont toutes deux des fonctions à nombre de paramètres variable. Elles peuvent donc être utilisées pour effectuer des écritures et des lectures multiples en un seul appel. Afin de leur permettre de déterminer la nature des données passées dans les arguments, elles attendent toutes les deux en premier paramètre une chaîne de caractères descriptive des arguments suivants. Cette chaîne est appelée *chaîne de format*, et elle permet de spécifier avec précision le type, la position et les options de format (précision, etc.) des données à traiter. Les deux sections suivantes décrivent la manière d'utiliser ces chaînes de format pour chacune des deux fonctions `printf` et `scanf`.

1.7.3. La fonction `printf`

La syntaxe générale de la fonction `printf` est la suivante :

```
printf(chaîne de format [, valeur [, valeur [...]]])
```

La chaîne de format peut contenir, comme on l'a au début de ce chapitre, du texte, mais elle est essentiellement employée pour décrire les types des paramètres suivants et le formatage qu'elle doit leur appliquer pour écrire leur valeur sur la sortie standard. La fonction `printf` peut donc afficher un nombre arbitraire de valeurs, la seule contrainte étant qu'il y ait le bon nombre de *formateurs* dans la chaîne de format. Si le nombre de paramètres est inférieur au nombre de valeurs à afficher, le programme plantera.

La fonction `printf` insère les valeurs associées aux formateurs au sein du texte de la chaîne de format. Les formateurs qu'elle contient sont tout simplement remplacés par la représentation textuelle des valeurs qui leurs sont associées. La fonction `printf` renvoie le nombre de caractères effectivement écrits.

Les formateurs utilisés pour la fonction `printf` peuvent être relativement complexes, surtout si l'on désire spécifier précisément le format des valeurs écrites. Nous allons donc dans un premier temps présenter les chaînes de format les plus simples qui permettent d'afficher les valeurs des types de base du langage.

Les formateurs commencent tous par le caractère de pourcentage (caractère '`%`'). Ce caractère peut être suivi de différentes options, et d'une lettre définissant le type de données de la valeur que le formateur doit afficher. Comme le caractère '`%`' est utilisé pour identifier les formateurs dans la chaîne de format, l'affichage de ce caractère se fait simplement en le doublant.

Les lettres utilisées pour les types de données de base du langage sont indiquées dans le tableau suivant :

Tableau 1-3. Chaînes de format de `printf` pour les types de base

Type de données	Caractère de formatage
Entier décimal signé	d
Entier décimal non signé	u ou i
Entier en octal	o

Type de données	Caractère de formatage
Entier en hexadécimal	x (avec les caractères 'a' à 'f') ou X (avec les caractères 'A' à 'F')
Flottants de type double	f, e, g, E ou G
Caractère isolé	c
Chaîne de caractères	s
Pointeur (voir Chapitre 4)	p

Vous noterez que la fonction `printf` n'est pas capable d'écrire des valeurs de type float. Elle ne peut travailler qu'avec des valeurs en virgule flottante de types double. Une conversion est effectuée automatiquement, il n'est donc pas nécessaire de la faire soi-même.

Les valeurs flottantes infinies sont remplacées par les mentions `+INF` et `-INF`. Un non-nombre (Not-A-Number) IEEE (norme utilisée pour la représentation des nombres en virgule flottate) donne `+NAN` ou `-NAN`.

Exemple 1-21. Affichage de valeurs simples

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    /* Définition de quelques variables : */
    int nb_notes = 3;
    double note_moyenne = 12.25;
    char classe = 'B';

    /* Exemples d'affichage de résultats : */
    printf("La moyenne des %d notes de la classe %c est %f.\n",
        nb_notes, classe, note_moyenne);
    printf("La meilleure appréciation est '%s'\n",
        "Bien");

    /* Exemple d'affichage du caractère % : */
    printf("%f%% des élèves sont admis\n", 78.3);

    /* Exemple d'écriture sur la sortie d'erreur standard : */
    fprintf(stderr, "Programme terminé avec succès !\n");
    return EXIT_SUCCESS;
}
```

En réalité, la syntaxe complètes des formateurs de `printf` est la suivante :

```
%[[indicateur]...][largeur][.précision][taille] type
```

Le champ `taille` permet de préciser la taille du type de données utilisé, si l'on n'utilise pas un type de données simple. Les champs `largeur` et `precision` permettent de contrôler la taille prise par la donnée formatée et sa précision. Enfin, le champ `indicateur` permet de spécifier le formatage du signe pour les nombres.

Les valeurs disponibles pour le paramètre de *taille* sont les caractères suivants :

Tableau 1-4. Options de taille pour les types dérivés

Option	Type de données
h	short int
l	long int ou wchar_t ou chaîne de caractères de type wchar_t
L	long double

Ainsi, pour écrire des caractères de type wchar_t et des chaînes de caractères Unicode, vous devrez préfixer respectivement les caractères de types '`c`' et '`s`' par la lettre '`l`'.

Exemple 1-22. Affichage de chaîne Unicode

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    printf("Ceci est une chaîne Unicode : '%ls'\n", L"Hello World!");
    return EXIT_SUCCESS;
}
```

Les valeurs disponibles pour le paramètre de *indicateur* sont les caractères suivants :

Tableau 1-5. Options d'alignements et de remplissange

Option	Signification
`0`	justification à droite de la sortie, avec remplissage à gauche par des 0.
-	justification à gauche de la sortie, avec remplissage à droite par des espaces.
+	affichage du signe pour les nombres positifs.
espace (' ')	les nombres positifs commencent tous par un espace.

Le champ *`largeur`* permet de spécifier la largeur minimale du champ de sortie. Si la sortie est trop petite, on complète avec des 0 ou des espaces selon l'indicateur utilisé. Notez qu'il s'agit bien d'une largeur minimale ici et non d'une largeur maximale. Le résultat du formatage de la donnée à écrire peut donc dépasser la valeur indiquée pour la largeur du champ.

Enfin, le champ *`précision`* spécifie la précision maximale de la sortie (nombre de chiffres à afficher pour les entiers, précision pour les flottants, et nombre de caractères pour les chaînes de caractères).

1.7.4. La fonction scanf

La fonction `scanf` permet de faire une ou plusieurs entrées. Comme la fonction `printf`, elle attend une chaîne de format en premier paramètre. Il faut ensuite passer les variables devant contenir les entrées dans les paramètres qui suivent. Sa syntaxe est la suivante :

```
scanf(chaîne de format, &variable [, &variable [...]]);
```

Elle renvoie le nombre de variables lues.

Ne cherchez pas à comprendre pour l'instant la signification du symbole `&` se trouvant devant chacune des variables. Sachez seulement que s'il est oublié, le programme plantera.

La chaîne de format peut contenir une chaînes de caractères et non uniquement des formateurs. Toutefois, si elle contient autre chose que des formateurs, le texte saisi par l'utilisateur devra impérativement correspondre avec la chaîne de caractères de la chaîne de format. `scanf` cherchera à reconnaître cette chaîne, et arrêtera l'analyse à la première erreur.

La syntaxe des formateurs pour `scanf` diffère un peu de celle de ceux de `printf` :

```
%[*][largeur][taille]type
```

Les types utilisés pour les chaînes de format de `scanf` sont semblables à ceux utilisés pour `printf`. Toutefois, on prendra garde au fait que le type '`f`' correspond cette fois bel et bien au type de données float et non au type de données double. L'analyse d'un double se fera donc en ajoutant le modificateur de taille '`l`'.

Le paramètre *`largeur`* permet quant à lui de spécifier le nombre maximal de caractères à prendre en compte lors de l'analyse du paramètre. Le paramètre '*' est facultatif, il indique seulement de passer la donnée entrée et de ne pas la stocker dans la variable destination. Attention, cette variable doit quand même être présente dans la liste des paramètres de `scanf`.

Exemple 1-23. Calcul de moyenne

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    double x, y;
    printf("Calcul de moyenne\n");   /* Affiche le titre. */
    printf("Entrez le premier nombre : ");
    scanf("%lf", &x);            /* Entre le premier nombre. */
    printf("\nEntrez le deuxième nombre : ");
    scanf("%lf", &y);            /* Entre le deuxième nombre. */
    printf("\nLa valeur moyenne de %f et de %f est %f.\n",
        x, y, (x+y)/2);
    return EXIT_SUCCESS;
}
```

Note : Vous noterez qu'il y a incohérence entre la chaîne de format utilisée pour les `scanf` et la chaîne de format pour le `printf`. Nul n'est parfait...

En pratique, la fonction `scanf` n'analyse les caractères provenant du flux d'entrée que lorsqu'une ligne complète a été saisie. Toutefois, elle ne supprime pas du tampon de flux d'entrée le caractère de saut de ligne, si bien qu'il s'y trouvera toujours lors de l'entrée suivante. Cela n'est pas gênant si l'on n'utilise que la fonction `scanf` pour réaliser les entrées de données dans le programme, car cette fonction ignore tout simplement ces caractères de saut de ligne. En revanche, si l'on utilise une autre fonction après un appel à `scanf`, il faut s'attendre à trouver ce caractère de saut de ligne dans le flux d'entrée.

La fonction `scanf` n'est pas très adaptée à la lecture des chaînes de caractères, car il n'est pas facile de contrôler la taille maximale que l'utilisateur peut saisir. C'est pour cette raison que l'on a généralement recours à la fonction `fgets`, qui permet de lire une ligne sur le flux d'entrée standard et de stocker le résultat dans une chaîne de caractères fournie en premier paramètre et dont la

longueur maximale est spécifiée en deuxième paramètre (nous verrons plus loin comment créer de telles chaînes de caractères). Le troisième paramètre de la fonction `fgets` est le flux à partir duquel la lecture de la ligne doit être réalisée, c'est à dire généralement `stdin`. L'analyse de la chaîne de caractères ainsi lue peut alors être faite avec une fonction similaire à la fonction `scanf`, mais qui lit les caractères à analyser dans une chaîne de caractères au lieu de les lire directement depuis le flux d'entrée standard : la fonction `sscanf`. Cette fonction s'utilise exactement comme la fonction `scanf`, à ceci près qu'il faut lui fournir en premier paramètre la chaîne de caractères dans laquelle se trouvent les données à interpréter. La description de ces deux fonctions dépasse le cadre de ce document et ne sera donc pas faite ici. Veuillez vous référer à la documentation de votre environnement de développement ou à la bibliographie pour plus de détails à leur sujet.

Chapitre 2. Les structures de contrôle

Nous allons aborder dans ce chapitre un autre aspect du langage indispensable à la programmation, à savoir : les *structures de contrôle*. Ces structures permettent, comme leur nom l'indique, de contrôler l'exécution du programme en fonction de critères particuliers. Le C et le C++ disposent de toutes les structures de contrôle classiques des langages de programmation comme les tests, les boucles, les sauts, etc. Toutes ces structures sont décrites dans les sections suivantes.

2.1. Les tests

Les tests sont les structures qui permettent de sélectionner une instruction ou un groupe d'instructions en fonction du résultat d'un test.

2.1.1. La structure conditionnelle if

La structure conditionnelle `if` permet de réaliser un test et d'exécuter une instruction ou non selon le résultat de ce test. Sa syntaxe est la suivante :

```
if (test) opération;
```

où `test` est une expression dont la valeur est booléenne ou entière. Toute valeur non nulle est considérée comme vraie. Si le test est vrai, `opération` est exécuté. Ce peut être une instruction ou un bloc d'instructions. Une variante permet de spécifier l'action à exécuter en cas de test faux :

```
if (test) opération1;
else opération2;
```

Note : Attention ! Les parenthèses autour du test sont nécessaires !

Les opérateurs de comparaison sont les suivants :

Tableau 2-1. Opérateurs de comparaison

`a == b`	Test d'égalité entre les expressions `a` et `b`
`a != b` ou `a not_eq b`	Test d'inégalité entre les expressions `a` et `b`
`a < b`	Test d'infériorité entre les expressions `a` et `b`
`a > b`	Test de supériorité entre les expressions `a` et `b`
`a <= b`	Test d'infériorité ou d'égalité entre les expressions `a` et `b`
`a >= b`	Test de supériorité ou d'égalité entre les expressions `a` et `b`

Les opérateurs logiques applicables aux expressions booléennes sont les suivants :

Tableau 2-2. Opérateurs logiques

`a && b` ou `a and b`	Conjonction des expressions booléennes `a` et `b` (« Et logique »)
`a || b` ou `a or b`	Disjonction des expressions booléennes `a` et `b` (« Ou logique »)
`!a` ou `not a`	Négation logique de l'expression booléenne `a`

Il n'y a pas d'opérateur ou logique exclusif.

Exemple 2-1. Test conditionnel if

```
if (a<b && a!=0)
{
    m=a;
    nouveau_m=1;
}
```

2.1.2. Le branchement conditionnel

Dans le cas où plusieurs instructions différentes doivent être exécutées selon la valeur d'une variable de type intégral, l'écriture de `if` successifs peut être relativement lourde. Le C/C++ fournit donc la structure de contrôle `switch`, qui permet de réaliser un branchement conditionnel. Sa syntaxe est la suivante :

```
switch (valeur)
{
case cas1:
    [instruction;
     [break;]
    ]
case cas2:
    [instruction;
     [break;]
    ]
    ⋮
case casN:
    [instruction;
     [break;]
    ]
[default:
    [instruction;
     [break;]
    ]
]
}
```

`valeur` est évalué en premier. Son type doit être entier. Selon le résultat de l'évaluation, l'exécution du programme se poursuit au cas de même valeur. Si aucun des cas ne correspond et si `default` est

présent, l'exécution se poursuit après `default`. Si en revanche `default` n'est pas présent, on sort du `switch`.

Les instructions qui suivent le `case` approprié ou `default` sont exécutées. Puis, les instructions *du cas suivant* sont également exécutées (on ne sort donc pas du `switch`). Pour forcer la sortie du `switch`, on doit utiliser le mot clé `break`.

Exemple 2-2. Branchement conditionnel switch

```
i= 2;
switch (i)
{
case 1:
case 2:    /* Si i=1 ou 2, la ligne suivante sera exécutée. */
    i=2-i;
    break;
case 3:
    i=0;   /* Cette ligne ne sera jamais exécutée. */
default:
    break;
}
```

Note : Il est interdit de définir une variable dans un des `case` d'un `switch`.

2.2. Les boucles

Les boucles sont des structures de contrôle qui permettent de réaliser une opération plusieurs fois, tant qu'une condition est vérifiée.

2.2.1. La boucle for

La structure de contrôle `for` est sans doute l'une des boucles les plus importantes. Elle permet de réaliser toutes sortes de boucles et, en particulier, les boucles permettant d'itérer sur un ensemble de valeur d'une variable de contrôle. Sa syntaxe est la suivante :

```
for (initialisation ; test ; itération) opération;
```

où `initialisation` est une instruction évaluée avant le premier parcours de la boucle du `for`. `test` est une expression dont la valeur déterminera la fin de la boucle. `itération` est une instruction effectuée à la fin de chaque passage dans la boucle, et `opération` constitue le traitement de la boucle elle-même. Chacune de ces parties est facultative.

La séquence d'exécution est la suivante :

```
initialisation
test
si vrai :
    opération
    itération
    retour au test
```

```
fin du for.
```

Exemple 2-3. Boucle for

```
somme = 0;
for (i=0; i<=10; i=i+1) somme = somme + i;
```

Note : En C++, il est possible que la partie `initialisation` déclare une variable. Dans ce cas, la variable déclarée n'est définie qu'à l'intérieur de l'instruction `for`. Par exemple,

```
for (int i=0; i<10; ++i);
```

est strictement équivalent à :

```
{
    int i;
    for (i=0; i<10; ++i);
}
```

Cela signifie que l'on ne peut pas utiliser la variable `i` après l'instruction `for`, puisqu'elle n'est définie que dans le corps de cette instruction. Cela permet de réaliser des variables muettes qui ne servent qu'à l'instruction `for` dans laquelle elles sont définies.

Note : Cette règle n'était pas respectée par certains compilateurs jusqu'à encore récemment. En effet, historiquement, les premiers compilateurs autorisaient bien la définition d'une variable dans la partie initialisation des boucles `for`, mais cette variable n'était pas limitée à l'intérieur de la boucle. Elle restait donc accessible après cette instruction. La différence est subtile, mais importante. Cela pose assurément des problèmes de compatibilité et les programmes C++ relativement anciens écrits pour ces compilateurs doivent être portés, puisque dans un cas la variable doit être redéclarée et dans l'autre cas elle ne le doit pas. Dans ce cas, le plus simple est de ne pas déclarer la variable de contrôle dans la partie `initialisation` de l'instruction `for`, mais juste avant, afin de revenir à la sémantique originelle.

2.2.2. Le while

Le `while` permet d'exécuter des instructions en boucle tant qu'une condition est vraie. Sa syntaxe est la suivante :

```
while (test) opération;
```

où `opération` est effectuée tant que `test` est vérifié. Comme pour le `if`, les parenthèses autour du test sont nécessaires. L'ordre d'exécution est :

```
test
si vrai :
    opération
    retour au test
```

Exemple 2-4. Boucle while

```
somme = i = 0;
while (somme<1000)
{
    somme = somme + 2 * i / (5 + i);
    i = i + 1;
}
```

2.2.3. Le do

La structure de contrôle `do` permet, tout comme le `while`, de réaliser des boucles en attente d'une condition. Cependant, contrairement à celui-ci, le `do` effectue le test sur la condition après l'exécution des instructions. Cela signifie que les instructions sont toujours exécutées au moins une fois, que le test soit vérifié ou non. Sa syntaxe est la suivante :

```
do opération;
while (test);
```

`opération` est effectuée jusqu'à ce que `test` ne soit plus vérifié.

L'ordre d'exécution est :

```
opération
test
si vrai, retour à opération
```

Exemple 2-5. Boucle do

```
p = i = 1;
do
{
    p = p * i;
    i = i + 1;
} while (i != 10);
```

2.3. Les instructions de rupture de séquence et de saut

Les instructions de rupture de séquence permettent, comme leur nom l'indique, d'interrompre une séquence d'instructions et de passer à la séquence suivante directement. Elles sont souvent utilisées pour sortir des boucles, pour passer à l'itération suivante, ou pour traiter une erreur qui s'est produite pendant un traitement.

2.3.1. Les instructions de rupture de séquence

Les commandes de *rupture de séquence* sont les suivantes :

```
return [valeur];

break;

continue;
```

`return` permet de quitter immédiatement la fonction en cours. Comme on l'a déjà vu, la commande `return` peut prendre en paramètre la valeur de retour de la fonction.

`break` permet de passer à l'instruction suivant l'instruction `while`, `do`, `for` ou `switch` la plus imbriquée (c'est-à-dire celle dans laquelle on se trouve).

`continue` saute directement à la dernière ligne de l'instruction `while`, `do` ou `for` la plus imbriquée. Cette ligne est l'accolade fermante. C'est à ce niveau que les tests de continuation sont faits pour `for` et `do`, ou que le saut au début du `while` est effectué (suivi immédiatement du test). On reste donc dans la structure dans laquelle on se trouvait au moment de l'exécution de `continue`, contrairement à ce qui se passe avec le `break`.

Exemple 2-6. Rupture de séquence par continue

```
/* Calcule la somme des 1000 premiers entiers pairs : */
somme_pairs=0;
for (i=0; i<1000; i=i+1)
{
    if (i % 2 == 1) continue;
    somme_pairs=somme_pairs + i;
}
```

2.3.2. Le saut

Le C/C++ dispose également d'une instruction de saut permettant de poursuivre l'exécution du programme en un autre point. Bien qu'il soit fortement déconseillé de l'utiliser, cette instruction est nécessaire et peut parfois être très utile, notamment dans les traitements d'erreurs. Sa syntaxe est la suivante :

```
goto étiquette;
```

où `étiquette` est une étiquette marquant la ligne destination dans la fonction. Les étiquettes sont simplement déclarées avec la syntaxe suivante :

```
étiquette:
```

Les étiquettes peuvent avoir n'importe quel nom d'identificateur.

Il n'est pas possible d'effectuer des sauts en dehors d'une fonction. En revanche, il est possible d'effectuer des sauts en dehors et à l'intérieur des blocs d'instructions sous certaines conditions. Si la destination du saut se trouve après une déclaration, cette déclaration ne doit pas comporter d'initialisations. De plus, ce doit être la déclaration d'un type simple (c'est-à-dire une déclaration qui ne demande pas l'exécution de code) comme les variables, les structures ou les tableaux. Enfin, si, au cours d'un saut, le contrôle d'exécution sort de la portée d'une variable, celle-ci est détruite.

Note : Ces dernières règles sont particulièrement importantes en C++ si la variable est un objet dont la classe a un constructeur ou un destructeur non trivial. Voir le Chapitre 7 pour plus de détails à ce sujet.

Autre règle spécifique au C++ : il est impossible d'effectuer un saut à l'intérieur d'un bloc de code en exécution protégée `try {}`. Voir aussi le Chapitre 8 concernant les exceptions.

Chapitre 3. Types avancés et classes de stockage

Nous avons présenté, dans le premier chapitre, les types fondamentaux du langage et leurs types dérivés. Nous allons maintenant pouvoir étudier plus en détails ces types. Nous verrons ensuite comment définir et utiliser des types plus complexes, tels que les tableaux, les structures et les unions. Nous verrons également comment simplifier l'utilisation des types de données et comment convertir des valeurs d'un type de données en un autre. Enfin, nous verrons les différentes possibilités pour spécifier la portée et la durée de vie des variables, via la notion de classe de stockage.

3.1. Types de données portables

La taille des types n'est spécifiée dans aucune norme. La seule chose qui est indiquée dans la norme C++, c'est que le plus petit type est le type char. Les tailles des autres types sont donc des multiples de celle du type char. De plus, les inégalités suivantes sont toujours vérifiées :

```
char ≤ short int ≤ int ≤ long int
float ≤ double ≤ long double
```

où l'opérateur « ≤ » signifie ici « a une plage de valeurs plus petite ou égale que ».

Cela dit, tous les environnements C/C++ stockent les char sur un octet, et le type short sur deux octets. Pour les autres types, ce n'est pas du tout aussi simple que cela...

La norme stipule que le type int est celui qui permet de stocker les entiers au format natif du processeur utilisé. Il doit donc être codé sur deux octets sur les machines 16 bits, sur quatre octets sur les machines 32 bits, et sur 8 octets sur les machines 64 bits. Cela implique que, sur les machines 64 bits, le type long est stocké lui aussi sur au moins 8 octets. De ce fait, il ne reste que le type de données short pour représenter les données 16 bits et 32 bits sur ces machines. De toutes évidences, il y a un problème, car on ne peut dans ce cas pas manipuler des données 16 bits ou 32 bits facilement.

Afin de résoudre ce problème, la plupart des compilateurs ne respectent tout simplement pas la norme ! Ils brisent en effet la règle selon laquelle le type int est le type des entiers natifs du processeur, et fixent sa taille à 32 bits quelle que soit l'architecture utilisée (sauf pour les vieilles architectures 16 bits, pour lesquelles la norme pouvait encore être respectée).

Ainsi, la taille des types utilisée *en pratique* est récapitulée dans le tableau suivant :

Type	**Architecture**		
	16 bits	**32 bits**	**64 bits**
char	8 bits	8 bits	8 bits
short	16 bits	16 bits	16 bits
int	16 bits	32 bits	32 bits
long	32 bits	32 bits	64 bits

La taille des caractères de type wchar_t quant à elle n'est pas spécifiée et dépend de l'environnement de développement utilisé. Ils sont généralement codés sur deux ou sur quatre octets suivant la représentation utilisée pour les caractères Unicode. Par exemple, ils sont codés sur deux octets sous

Windows, et sur quatre sous Unix.

Là où les choses se compliquent, c'est que les valeurs accessibles sont fortement dépendantes du nombre de bits utilisés pour les stocker. Elles dépendent également du signe du type de données. En effet, pour les types signés, un bit est utilisé pour stocker l'information de signe de la valeur. De ce fait, la plage de valeurs utilisables en valeur absolue est moins grande.

Par exemple, si le type char est codé sur 8 bits, on peut coder les nombres allant de 0 à 255 avec ce type en non signé (il y a 8 chiffres binaires, chacun peut valoir 0 ou 1, on a donc 2 puissance 8 combinaisons possibles, ce qui fait 256). En signé, les valeurs s'étendent de -128 à 127 (un des chiffres binaires est utilisé pour le signe, il en reste 7 pour coder le nombre, donc il reste 128 possibilités dans les positifs comme dans les négatifs. 0 est considéré comme positif. En tout, il y a autant de possibilités.). De même, si le type int est codé sur 16 bits (cas des machines 16 bits), les valeurs accessibles vont de -32768 à 32767 ou de 0 à 65535 si l'entier n'est pas signé. C'est le cas sur les PC en mode réel (c'est-à-dire sous DOS) et sous Windows 3.x. Sur les machines fonctionnant en 32 bits, le type int est stocké sur 32 bits : l'espace des valeurs disponibles est donc 65536 fois plus large. C'est le cas sur les PC en mode protégé 32 bits (Windows 9x, ou basés sur la technologie NT, DOS Extender, Linux) et sur les Macintosh. C'est aussi le cas sur la plupart des machines 64 bits, pour les raisons que l'on a vues.

On constate donc que la portabilité des types de base est très aléatoire. Cela signifie qu'il faut faire extrêmement attention dans le choix des types si l'on veut faire du code portable (c'est-à-dire qui compilera et fonctionnera sans modifications du programme sur tous les ordinateurs). Il est dans ce cas nécessaire d'utiliser des types de données qui donnent les mêmes intervalles de valeurs sur tous les ordinateurs.

Afin de régler tous ces problèmes, la norme ISO C99 impose de définir des types portables afin de régler ces problèmes sur toutes les architectures existantes. Ces types sont définis dans le fichier d'entête `stdint.h`. Il s'agit des types int8_t, int16_t, int32_t et int64_t, et de leurs versions non signées uint8_t, uint16_t, uint32_t et uint64_t. La taille de ces types en bits est indiquée dans leur nom et leur utilisation ne devrait pas poser de problème.

> **Note :** Le problème se pose également pour les types flottants, mais il est bien moins grave parce que la plupart des compilateurs utilisent les formats IEEE normalisés pour les nombres en virgule flottante. Le type float est généralement codé sur 4 octets, et les types double et long double sont souvent identiques et codés sur 8 octets. En pratique, le type float est toutefois très souvent trop restreint pour être utilisable de nos jours, et on lui préférera le type double.
>
> Le programmeur devra également faire attention, en plus du problème de la taille des types, au problème de la représentation des données. En effet, deux représentations d'un même type peuvent être différentes en mémoire sur deux machines d'architectures différentes, même à taille égale en nombre de bits. Le problème le plus courant est l'ordre de stockage des octets en mémoire pour les types qui sont stockés sur plus d'un octet (c'est-à-dire quasiment tous). Certaines machines stockent les octets de poids fort en premier, d'autres les octets de poids faible. Ce problème revêt donc une importance capitale lorsque le programme doit être portable ou que des données doivent être échangées entre des machines d'architectures a priori différentes, par exemple dans le cadre d'une communication réseau ou lors d'un échange de fichier. Une solution simple est de toujours échanger les données au format texte (moyennant les précautions nécessaires pour prendre en charge les erreurs de conversion induites par le formatage), ou de choisir un mode de représentation de référence. Les bibliothèques réseau disposent généralement des méthodes permettant de convertir les données vers un format commun d'échange de données par un réseau et pourront par exemple être utilisées.

3.2. Structures de données et types complexes

En dehors des types de variables simples, le C/C++ permet de créer des types plus complexes. Ces types comprennent essentiellement les tableaux, les structures, les unions et les énumérations.

3.2.1. Les tableaux

La définition d'un tableau se fait en faisant suivre le nom de l'identificateur d'une paire de crochets, contenant le nombre d'élément du tableau :

```
type identificateur[taille]([taille](...));
```

> **Note :** Attention ! Les caractères '`[`' et '`]`' étant utilisés par la syntaxe des tableaux, ils ne signifient plus les éléments facultatifs ici. Ici, et ici seulement, les éléments facultatifs sont donnés entre parenthèses.

Dans la syntaxe précédente, type représente le type des éléments du tableau.

Exemple 3-1. Définition d'un tableau

```
int MonTableau[100];
```

`MonTableau` est un tableau de 100 entiers. On référence les éléments des tableaux en donnant l'indice de l'élément entre crochet :

```
MonTableau[3]=0;
```

Les indices des tableaux varient de `0` à `taille-1`. Il y a donc bien `taille` éléments dans le tableau. Dans l'exemple donné ci-dessus, l'élément `MonTableau[100]` n'existe pas : y accéder plantera le programme. C'est au programmeur de vérifier que ses programmes n'utilisent jamais les tableaux avec des indices négatifs ou plus grands que leur taille.

En C/C++, les tableaux à plus d'une dimension sont des tableaux de tableaux. On prendra garde au fait que dans la définition d'un tableau à plusieurs dimensions, la dernière taille indiquée spécifie la taille du tableau dont on fait un tableau. Ainsi, dans l'exemple suivant :

```
int Matrice[5][4];
```

`Matrice` est un tableau de taille 5 dont les éléments sont eux-mêmes des tableaux de taille 4. L'ordre de déclaration des dimensions est donc inversé : 5 est la taille de la dernière dimension et 4 est la taille de la première dimension. L'élément suivant :

```
Matrice[2];
```

est donc le troisième élément de ce tableau de taille cinq, et est lui-même un tableau de quatre éléments.

Il est possible, lors d'une *déclaration* de tableau (pas lors d'une définition), d'omettre la taille de la dernière dimension d'un tableau (donc la taille du premier groupe de crochets). En effet, le C/C++ ne contrôle pas la taille des tableaux lui-même et, pour lui, un tableau peut avoir une taille quelconque.

Seul le type de données des éléments est important pour calculer leur taille et donc la position de chacun d'eux en mémoire par rapport au début du tableau. De ce fait, la taille de la dernière dimension n'est pas nécessaire dans la déclaration, elle n'est en fait utile que lors de la définition pour réserver effectivement la mémoire du tableau.

Cette écriture peut par exemple être utilisée pour passer des tableaux en paramètre à une fonction.

Exemple 3-2. Passage de tableau en paramètre

```
void f(int taille, int t[][20])
{
    /* Utilisation de t[i][j] ... */
    return;
}

int main(void)
{
    int tab[10][20];
    test(10, tab);   /* Passage du tableau en paramètre. */
    return 0;
}
```

Dans cet exemple, la fonction `f` reçoit un tableau à deux dimensions en paramètre, dont seule la première dimension est spécifiée. Bien entendu, pour qu'elle puisse l'utiliser correctement, elle doit en connaître la taille, donc la taille de la deuxième dimension. Cela peut se faire soit en passant un paramètre explicitement comme c'est le cas ici, ou en utilisant une convention dont le programmeur doit s'assurer lors de l'appel de la fonction.

> **Note :** Attention, les tableaux sont passés par référence dans les fonctions. Cela signifie que si la fonction modifie les éléments du tableau, elle modifie les éléments du tableau que l'appelant lui a fourni. Autrement dit, il n'y a pas de copie du tableau lors de l'appel de la fonction.
>
> Les tailles des premières dimensions sont absolument nécessaires. Outre le fait qu'elles permettent de déterminer la taille de chaque élément de la dernière dimension, elle permettent également au compilateur de connaître le rapport des dimensions entre elles. Par exemple, la syntaxe :
>
> ```
> int tableau[][];
> ```
>
> utilisée pour référencer un tableau de 12 entiers ne permettrait pas de faire la différence entre les tableaux de deux lignes et de six colonnes et les tableaux de trois lignes et de quatre colonnes (et leurs transposés respectifs). Une référence telle que :
>
> ```
> tableau[1][3]
> ```
>
> ne représenterait rien. Selon le type de tableau, l'élément référencé serait le quatrième élément de la deuxième ligne (de six éléments), soit le dixième élément, ou bien le quatrième élément de la deuxième ligne (de quatre éléments), soit le huitième élément du tableau. En précisant tous les indices sauf un, il est possible de connaître la taille du tableau pour cet indice à partir de la taille globale du tableau, en la divisant par les tailles sur les autres dimensions (2 = 12/6 ou 3 = 12/4 par exemple).

3.2.2. Les chaînes de caractères

Comme on l'a déjà dit, il n'existe pas de type de données spécifique pour manipuler les chaînes de caractères en C/C++. Les chaînes de caractères sont en effet considérées comme des tableaux de caractères, dont le dernier caractère est nul. Ce carctère permet de marquer la fin de la chaîne de caractères et ne devra jamais être oublié.

De ce fait, les tableaux de caractères sont très utilisés pour stocker des chaînes de caractères. Il faudra toutefois faire très attention à toujours utiliser des tailles de tableaux d'une unité supérieures à la taille des chaînes de caractères à stocker, afin de pouvoir placer le caractère nul terminal de la chaîne. Par exemple, pour créer une chaîne de caractères de 12 caractères au plus, il faut un tableau pour 13 caractères.

Exemple 3-3. Chaîne de caractères C et tableau

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    /* Réserve assez de place pour
       la chaîne et son caractère nul terminal : */
    char chaine[13];

    /* Copie la chaîne dans le tableau : */
    strcpy(chaine, "Hello World!");

    /* Affiche la chaîne : */
    printf("%s\n", chaine);    /* Affiche "Hello World!" */

    /* Tronque la chaîne avec un autre nul : */
    chaine[5] = '\0';
    printf("%s\n", chaine);    /* Affiche "Hello" */

    /* Insère un caractère '_' : */
    chaine[5] = '_';
    printf("%s\n", chaine);    /* Affiche "Hello_World!" */

    return EXIT_SUCCESS;
}
```

Cet exemple montre bien que le caractère nul marque la fin la chaîne de caractère pour les fonctions de la bibliothèque C. Cela n'implique pas que les caractères situés après ce caractère nul sont détruits, mais que du point de vue de la bibliothèque, la chaîne s'arrête au niveau de ce caractère.

> **Note :** La fonction `strcpy` utilisée dans cet exemple est l'une des nombreuses fonctions de manipulation de chaînes de caractères de la bibliothèque C. Elle permet de réaliser la copie d'une chaîne de caractères dans une autre (attention, elle ne vérifie pas que la taille de la chaîne cible est suffisante). Ces fonctions sont déclarées dans le fichier d'en-tête `string.h`.

3.2.3. Les structures

Les *structures* sont des aggrégats de données de types plus simples. Les structures permettent de construire des types complexes à partir des types de base ou d'autres types complexes.

La définition d'une structure se fait à l'aide du mot clé `struct` :

```
struct [nom_structure]
{
    type champ;
   [type champ;
   [...]]
};
```

La structure contient plusieurs données membres, appelées *champs*. Leurs types sont donnés dans la déclaration de la structure. Ces types peuvent être n'importe quel autre type, même une structure. Le nom de la structure est facultatif et peut ne pas être précisé dans les situations où il n'est pas nécessaire (voir plus loin).

La structure ainsi définie peut alors être utilisée pour définir une variable dont le type est cette structure.

Pour cela, deux possibilités :

- faire suivre la définition de la structure par l'identificateur de la variable ;

  Exemple 3-4. Déclaration de variable de type structure

  ```
  struct Client
  {
      unsigned char Age;
      unsigned char Taille;
  } Jean;
  ```

  ou, plus simplement :

  ```
  struct
  {
      unsigned char Age;
      unsigned char Taille;
  } Jean;
  ```

  Dans le deuxième exemple, le nom de la structure n'est pas mis.

- déclarer la structure en lui donnant un nom, puis déclarer les variables avec la syntaxe suivante :

  ```
  [struct] nom_structure identificateur;
  ```

  Exemple 3-5. Déclaration de structure

  ```
  struct Client
  {
      unsigned char Age;
      unsigned char Taille;
  };
  ```

```
struct Client Jean, Philippe;
Client Christophe;   // Valide en C++ mais invalide en C
```

Dans cet exemple, le nom de la structure doit être mis, car on utilise cette structure à la ligne suivante. Pour la déclaration des variables `Jean` et `Philippe` de type struct Client, le mot clé `struct` a été mis. Cela n'est pas nécessaire en C++, mais l'est en C. Le C++ permet donc de déclarer des variables de type structure exactement comme si le type structure était un type prédéfini du langage. La déclaration de la variable `Christophe` ci-dessus est invalide en C.

Les éléments d'une structure sont accédés par un point, suivi du nom du champ de la structure à accéder. Par exemple, l'âge de `Jean` est désigné par `Jean.Age`.

Il est possible de ne pas donner de nom à une structure lors de sa définition sans pour autant déclarer une variable. De telles structures anonymes ne sont utilisables que dans le cadre d'une structure incluse dans une autre structure :

```
struct struct_principale
{
    struct
    {
        int champ1;
    };
    int champ2;
};
```

Dans ce cas, les champs des structures imbriquées seront accédés comme s'il s'agissait de champs de la structure principale. La seule limitation est que, bien entendu, il n'y ait pas de conflit entre les noms des champs des structures imbriquées et ceux des champs de la structure principale. S'il y a conflit, il faut donner un nom à la structure imbriquée qui pose problème, en en faisant un vrai champ de la structure principale.

3.2.4. Les unions

Les *unions* constituent un autre type de structure. Elles sont déclarées avec le mot clé `union`, qui a la même syntaxe que `struct`. La différence entre les structures et les unions est que les différents champs d'une union occupent le même espace mémoire. On ne peut donc, à tout instant, n'utiliser qu'un des champs de l'union.

Exemple 3-6. Déclaration d'une union

```
union entier_ou_reel
{
    int entier;
    float reel;
};

union entier_ou_reel x;
```

`x` peut prendre l'aspect soit d'un entier, soit d'un réel. Par exemple :

```
x.entier=2;
```

affecte la valeur `2` à `x.entier`, ce qui détruit `x.reel`.

Si, à présent, on fait :

```
x.reel=6.546;
```

la valeur de `x.entier` est perdue, car le réel `6.546` a été stocké au même emplacement mémoire que l'entier `x.entier`.

Les unions, contrairement aux structures, sont assez peu utilisées, sauf en programmation système où l'on doit pouvoir interpréter des données de différentes manières selon le contexte. Dans ce cas, on aura avantage à utiliser des unions de structures anonymes et à accéder aux champs des structures, chaque structure permettant de manipuler les données selon une de leurs interprétations possibles.

Exemple 3-7. Union avec discriminant

```
struct SystemEvent
{
    int iEventType;  /* Discriminant de l'événement.
                        Permet de choisir comment l'interpréter. */
    union
    {
        struct
        {                       /* Structure permettant d'interpréter */
            int iMouseX;        /* les événements souris. */
            int iMouseY;
        };
        struct
        {                       /* Structure permettant d'interpréter */
            char cCharacter;    /* les événements clavier. */
            int iShiftState;
        };
        /* etc. */
    };
};

/* Exemple d'utilisation des événements : */
int ProcessEvent(struct SystemEvent e)
{
    int result;
    switch (e.iEventType)
    {
    case MOUSE_EVENT:
        /* Traitement de l'événement souris... */
        result = ProcessMouseEvent(e.iMouseX, e.iMouseY);
        break;
    case KEYBOARD_EVENT:
        /* Traitement de l'événement clavier... */
        result = ProcessKbdEvent(e.cCharacter, e.iShiftState);
        break;
    }
    return result;
}
```

3.2.5. Les champs de bits

Il est possible de définir des structures dont les champs ne sont stockés que sur quelques bits et non sur des types de base du langage complets. De telles structures sont appelées des « *champs de bits* ».

Les champs de bits se déclarent comme des structures, avec le mot clé `struct`, mais dans lesquelles la taille en bits de chaque champ est spécifiée. Les différents groupes de bits doivent être consécutifs.

Exemple 3-8. Déclaration d'un champs de bits

```
struct champ_de_bits
{
    int var1;                    /* Définit une variable classique. */
    int bits1a4  : 4;            /* Premier champ : 4 bits. */
    int bits5a10 : 6;            /* Deuxième champ : 6 bits. */
    unsigned int bits11a16 : 6;  /* Dernier champ : 6 bits. */
};
```

La taille d'un champ de bits ne doit pas excéder celle d'un entier. Pour aller au-delà, on créera un deuxième champ de bits. La manière dont les différents groupes de bits sont placés en mémoire dépend du compilateur et n'est pas normalisée.

Les différents bits ou groupes de bits seront tous accessibles comme des variables classiques d'une structure ou d'une union :

```
struct champ_de_bits essai;

int main(void)
{
    essai.bits1a4 = 3;
    /* suite du programme */
    return 0;
}
```

3.2.6. Initialisation des structures et des tableaux

Les tableaux et les structures peuvent être initialisés dès leur création, tout comme les types classiques peuvent l'être. La valeur servant à l'initialisation est décrite en mettant les valeurs des membres de la structure ou du tableau entre accolades et en les séparant par des virgules :

Exemple 3-9. Initialisation d'une structure

```
/* Définit le type Client : */
struct Client
{
    unsigned char Age;
    unsigned char Taille;
    unsigned int Comptes[10];
};

/* Déclare et initialise la variable John : */
struct Client John={35, 190, {13594, 45796, 0, 0, 0, 0, 0, 0, 0, 0}};
```

La variable `John` est ici déclarée comme étant de type Client et initialisée comme suit : son âge est de `35`, sa taille de `190` et ses deux premiers comptes de `13594` et `45796`. Les autres comptes sont nuls.

Il n'est pas nécessaire de respecter l'imbrication du type complexe au niveau des accolades, ni de fournir des valeurs d'initialisations pour les derniers membres d'un type complexe. Les valeurs par défaut qui sont utilisées dans ce cas sont les valeurs nulles du type du champ non initialisé. Ainsi, la déclaration de `John` aurait pu se faire ainsi :

```
struct Client John={35, 190, 13594, 45796};
```

La norme C99 du langage C fournit également une autre syntaxe plus pratique pour initialiser les structures. Cette syntaxe permet d'initialiser les différents champs de la structure en les nommant explicitement et en leur affectant directement leur valeur. Ainsi, avec cette nouvelle syntaxe, l'initialisation précédente peut être réalisée de la manière suivante :

Exemple 3-10. Initialisation de structure C99

```
/* Déclare et initialise la variable John : */
struct Client John={
    .Taille = 190,
    .Age = 35,
    .Comptes[0] = 13594,
    .Comptes[1] = 45796
};
```

On constatera que les champs qui ne sont pas explicitement initialisés sont, encore une fois, initialisés à leur valeur nulle. De plus, comme le montre cet exemple, il n'est pas nécessaire de respecter l'ordre d'apparition des différents champs dans la déclaration de la structure pour leur initialisation.

Il est possible de mélanger les deux syntaxes. Dans ce cas, les valeurs pour lesquelles aucun nom de champ n'est donné seront affectées au champs suivants le dernier champ nommé. De plus, si plusieurs valeurs différentes sont affectées au même champ, seule la dernière valeur indiquée sera utilisée.

Cette syntaxe est également disponible pour l'initialisation des tableaux. Dans ce cas, on utilisera les crochets directement, sans donner le nom du tableau (exactement comme l'initialisation des membres de la structure utilise directement le point, sans donner le nom de la structure en cours d'initialisation).

Exemple 3-11. Initialisation de tableau C99

```
/* Déclare et initialise un tableau d'entier : */
int tableau[6]={
    [0] = 0,
    [1] = 2,
    [5] = 7
};
```

> **Note :** La syntaxe d'initialisation C99 n'est pas disponible en C++. Avec ce langage, il est préférable d'utiliser la notion de classe et de définir un constructeur. Les notions de classe et de constructeur seront présentées plus en détails dans le Chapitre 7. C'est l'un des rares points syntaxiques où il y a incompatibilité entre le C et le C++.

3.3. Les énumérations

Les *énumérations* sont des types *intégraux* (c'est-à-dire qu'ils sont basés sur les entiers), pour lesquels chaque valeur dispose d'un nom unique. Leur utilisation permet de définir les constantes entières dans un programme et de les nommer. La syntaxe des énumérations est la suivante :

```
enum enumeration
{
    nom1 [=valeur1]
    [, nom2 [=valeur2]
    [...]]
};
```

Dans cette syntaxe, `enumeration` représente le nom de l'énumération et `nom1`, `nom2`, etc. représentent les noms des énumérés. Par défaut, les énumérés reçoivent les valeurs entières `0`, `1`, etc. sauf si une valeur explicite leur est donnée dans la déclaration de l'énumération. Dès qu'une valeur est donnée, le compteur de valeurs se synchronise avec cette valeur, si bien que l'énuméré suivant prendra pour valeur celle de l'énuméré précédent augmentée de `1`.

Exemple 3-12. Déclaration d'une énumération

```
enum Nombre
{
    un=1, deux, trois, cinq=5, six, sept
};
```

Dans cet exemple, les énumérés prennent respectivement leurs valeurs. Comme `quatre` n'est pas défini, une resynchronisation a lieu lors de la définition de `cinq`.

Les énumérations suivent les mêmes règles que les structures et les unions en ce qui concerne la déclaration des variables : on doit répéter le mot clé `enum` en C, ce n'est pas nécessaire en C++.

3.4. Les alias de types

Le C/C++ dispose d'un mécanisme de création d'*alias* et de synonymes d'autres types. Ce mécanisme permet souvent de simplifier les écritures, mais peut aussi être utile lors de la création de nouveaux types.

3.4.1. Définition d'un alias de type

La définition d'un alias de type se fait à l'aide du mot clé `typedef`. Sa syntaxe est la suivante :

```
typedef définition alias;
```

où `alias` est le nom que doit avoir le synonyme du type et `définition` est sa définition.

Exemple 3-13. Définition de type simple

```
typedef unsigned int mot;
```

mot est alors strictement équivalent à unsigned int.

Pour les tableaux, la syntaxe est particulière :

```
typedef type_tableau type[(taille)]([taille](...));
```

`type_tableau` est alors le type des éléments du tableau.

Exemple 3-14. Définition de type tableau

```
typedef int tab[10];
```

tab est le synonyme de « tableau de 10 entiers ».

La définition d'alias de type est très utile pour donner un nom aux structures et ainsi simplifier leur utilisation en C.

Exemple 3-15. Définition de type structure

```
typedef struct client
{
    unsigned int Age;
    unsigned int Taille;
} Client;
```

Client représente la structure struct client. Le nom de l'alias (« Client ») pourra dès lors être utilisé en lieu et place du nom de la structure (« struct client »).

> **Note :** Pour comprendre la syntaxe de `typedef`, il suffit de raisonner de la manière suivante. Si l'on dispose d'une expression qui permet de déclarer une variable d'un type donné, alors il suffit de placer le mot clé `typedef` devant cette expression pour faire en sorte que l'identificateur de la variable devienne un identificateur de type. Par exemple, si on supprime le mot clé `typedef` dans la déclaration du type Client ci-dessus, alors Client devient une variable dont le type est struct client.

3.4.2. Utilisation d'un alias de type

Les alias de types peuvent être utilisés exactement comme des types normaux. Il suffit de les utiliser en lieu et place des types qu'ils représentent. Par exemple, avec les alias de types définis précédemment, on pourrait écrire :

```
unsigned int i = 2, j;   /* Déclare deux unsigned int */
tab Tableau;   /* Déclare un tableau de 10 entiers */
Client John;   /* Déclare une structure client */

John.Age = 35; /* Initialise la variable John */
John.Taille = 175;
for (j=0; j<10; j = j+1) Tableau[j]=j;  /* Initialise Tableau */
```

Attention toutefois. Les alias de types constituent une facilité d'écriture, rien de plus. Ils ne permettent pas, contrairement à ce que l'éthymologie du mot clé `typedef` pourrait laisser penser, de créer de nouveaux types de données. Les alias de types sont bel et bien considérés comme étant identique aux types qu'ils représentent par le compilateur.

Cela signifie que le compilateur ne fera aucune vérification de types lors de la manipulation de types synonymes, et qu'il ne les distinguera pas non plus dans les signatures des fonctions. Par conséquent, il n'est pas possible de définir deux fonctions surchargées dont les paramètres ne diffèrent que par un alias de type.

Toutefois, le fait que l'on puisse créer un nouveau type en définissant une structure peut être utilisé avantageusement dans de nombreuses situations en conjonction de la définition d'un nouveau type. La création d'un type de données par l'intermédiaire d'une structure peut en effet être utilisée dans d'autres situations que la simple création d'un aggrégat de données de types plus simples.

En effet, il est relativement courant que des données aient le même type d'un point de vue représentation des données, mais que leurs types effectifs n'aient pas la même sémantique (chose que seul le programmeur peut savoir, et pour laquelle le compilateur ne peut être d'aucune aide).

Par exemple, un programme de conversion d'unités de mesures pourrait fort bien avoir besoin de manipuler des mesures de longueur exprimées en centimètres et d'autres en pouces. Il peut stocker ces différentes valeurs dans des variables de type entière, mais cela n'est pas recommandé, car il serait alors très facile de faire des calculs mixant les deux unités de mesures.

Dans ce genre de situation, il peut être utile d'exploiter les mécanismes de typage du langage. Comme `typedef` ne permet pas de sous-typer le type int en deux types distincts, la solution consiste dans ce cas à encapsuler le type de base int dans deux structures, et de définir ainsi deux types distincts pour chacune des unités de mesures :

```
typedef struct { int cm; } centimetres_t;
typedef struct { int inches; } pouces_t;
```

Une fois cela réalisé, des variables de type centimetre_t et pouces_t pourront être définies et utilisées sans risque de mélange involontaire. En effet, l'affectation d'une variable d'un type à une de l'autre type provoquera immédiatement une erreur de compilation.

Bien entendu, cela se fera au détriment de la simplicité d'écriture sur les opérations, puisqu'il faudra spécifier la donnée membre `cm` ou la données membre `inches` à chaque accès à la valeur du type de données. Mais la fiabilité du programme sera nettement supérieure :

```
centimetres_t l1, l2, l3;
pouces_t l4;
l1.cm = 12;
l2 = l1;              /* OK. */
l4 = l3;              /* Erreur, les pouces et les centimètres sont incompatibles ! */
l3.cm = l1.cm * 2;    /* OK. */
l4.cm = l2.cm + 1;    /* Erreur, les pouces n'ont pas la donnée membre cm ! */
```

Nous verrons toutefois que le C++ permet de résoudre ce problème en redéfinissant les opérateurs sur les types de données. Ceux-ci seront alors utilisables directement, comme des types de données natifs (voir la Section 7.11).

3.5. Transtypages et promotions

Il est parfois utile de changer le type d'une valeur. Considérons l'exemple suivant : la division de `5` par `2` renvoie `2`. En effet, `5/2` fait appel à la division euclidienne. Comment faire pour obtenir le résultat

avec un nombre réel ? Il faut faire `5./2`, car alors `5.` est un nombre flottant. Mais que faire quand on se trouve avec des variables entières (`i` et `j` par exemple) ? Le compilateur signale une erreur après `i` dans l'expression `i./j` ! Il faut changer le type de l'une des deux variables. Cette opération s'appelle le *transtypage*. On la réalise simplement en faisant précéder l'expression à transtyper du type désiré entouré de parenthèses :

```
(type) expression
```

Exemple 3-16. Transtypage en C

```
int i=5, j=2;
((float) i)/j
```

Dans cet exemple, `i` est transtypé en flottant avant la division. On obtient donc `2.5`.

Le transtypage C est tout puissant et peut être relativement dangereux. Le langage C++ fournit donc des opérateurs de transtypages plus spécifiques, qui permettent par exemple de conserver la constance des variables lors de leur transtypage. Ces opérateurs seront décrits dans la Section 9.2 du chapitre traitant de l'identification dynamique des types.

Le compilateur peut également effectuer des transtypages automatiquement, notamment lors d'une évaluation d'expression ou lors du passage d'un paramètre à une fonction qui attend une valeur d'un autre type. Généralement, le compilateur ne réalise de tels transtypages automatiques que lorsque le type de données cible est considéré comme plus grand que le type de données source. Par exemple, la conversion implicite d'un entier en flottant est généralement réalisée de manière transparente, alors que la conversion inverse provoque un avertissement. Mais attention, ce n'est pas toujours le cas. Par exemple, le compilateur convertira automatiquement, et sans avertissement, un entier en caractère, ou un nombre d'un type non signé vers le type correspondant signé. Cela peut conduire à des bogues assez graves.

> **Note :** De ce fait, le choix du type des variables devra se faire en toute connaissance de cause. De manière générale, il est conseillé de ne pas chercher à gagner des bouts de chandelles en réduisant la taille des types de données, d'une part, et de se baser sur la sémantique des valeurs manipulées, d'autre part. Par exemple, la taille d'un objet ne peut pas être négative, et devra généralement être manipulée via un type de données non signé. Inversement, une différence entre deux valeurs sera souvent signée (sauf si l'ordre des opérandes est choisi de telle manière que cela ne soit pas le cas).

Enfin, le compilateur peut effectuer des *promotions* sur les données dans certaines circonstances. Une promotion est un transtypage vers un type de données de taille supérieure, donc, a priori, sans perte de données. C'est en particulier le cas lors de l'appel des fonctions à nombre d'arguments variables. Ces fonctions n'acceptent pas n'importe quel type de données pour leurs arguments. En particulier, les types char et short ne sont pas utilisés : les paramètres sont toujours promus aux type int ou long int.

Les règles de promotion utilisées sont les suivantes :

- les types char, signed char, unsigned char, short int ou unsigned short int sont promus en int si ce type est capable d'accepter toutes leurs valeurs sur la plateforme considérée. Si int est insuffisant, unsigned int est utilisé ;

- les types des énumérations et wchar_t sont promus en int, unsigned int, long ou unsigned long selon leurs capacités. Le premier type capable de conserver la plage de valeur du type à promouvoir est utilisé ;
- les valeurs des champs de bits sont converties en int ou unsigned int selon la taille du champ de bit ;
- les valeurs de type float sont converties en double.

3.6. Les classes de stockage

Les variables C/C++ peuvent être créées de différentes manières. Il est courant, selon la manière dont elles sont créées et la manière dont elles pourront être utilisées, de les classer en différentes catégories de variables.

La classification la plus simple que l'on puisse faire des variables est la classification locale - globale. Les variables *globales* sont déclarées en dehors de tout bloc d'instructions, dans la zone de déclaration globale du programme. Les variables *locales* en revanche sont créées à l'intérieur d'un bloc d'instructions. Les variables locales et globales ont des durées de vie, des portées et des emplacements en mémoire différents.

La *portée* d'une variable est la zone du programme dans laquelle elle est accessible. La portée des variables globales est tout le programme, alors que la portée des variables locales est le bloc d'instructions dans lequel elles ont été créées.

La *durée de vie* d'une variable est le temps pendant lequel elle existe. Les variables globales sont créées au début du programme et détruites à la fin, leur durée de vie est donc celle du programme. En général, les variables locales ont une durée de vie qui va du moment où elles sont déclarées jusqu'à la sortie du bloc d'instructions dans lequel elles ont été déclarées. Cependant, il est possible de faire en sorte que les variables locales survivent à la sortie de ce bloc d'instructions, et soient à nouveau utilisables dès que l'on y entre à nouveau.

> **Note :** La portée d'une variable peut commencer avant sa durée de vie si cette variable est déclarée après le début du bloc d'instructions dans lequel elle est déclarée. La durée de vie n'est donc pas égale à la portée d'une variable.

La *classe de stockage* d'une variable permet de spécifier sa *durée de vie* et sa *place en mémoire* (sa portée est toujours le bloc dans lequel la variable est déclarée). Le C/C++ dispose d'un éventail de classes de stockage assez large et permet de spécifier le type de variable que l'on désire utiliser :

Tableau 3-1. Classes de stockages

Classe	Signification
`auto`	Classe de stockage par défaut. Les variables ont pour portée le bloc d'instructions dans lequel elles ont été crées. Elles ne sont accessibles que dans ce bloc. Leur durée de vie est restreinte à ce bloc.

Classe	Signification
`static`	Classe de stockage permettant de créer des variables dont la portée est le bloc d'instructions en cours, mais qui, contrairement aux variables `auto`, ne sont pas détruites lors de la sortie de ce bloc. À chaque fois que l'on rentre dans ce bloc d'instructions, les variables statiques existeront et auront pour valeurs celles qu'elles avaient avant que l'on quitte ce bloc. Leur durée de vie est donc celle du programme, et elles conservent leurs valeurs. Un fichier peut être considéré comme un bloc. Ainsi, une variable statique d'un fichier ne peut pas être accédée à partir d'un autre fichier. Cela est utile pour isoler des variables globales d'un fichier vis à vis des autres modules (voir le Chapitre 6 pour plus de détails).
`extern`	Classe de stockage utilisée dans les déclarations pour signaler que la variable ainsi déclarée peut être définie dans un autre fichier que le fichier courant. Elle est utilisée dans le cadre de la compilation séparée (voir le Chapitre 6 pour plus de détails).
`register`	Classe de stockage permettant de créer une variable dont l'emplacement se trouve dans un registre du microprocesseur. Il faut bien connaître le langage machine pour correctement utiliser cette classe de variable. En pratique, cette classe est très peu utilisée, et on péfère de nos jours laisser le compilateur gérer lui-même l'usage des registres du processeur.
`volatile`	Cette classe de stockage sert lors de la programmation système. Elle indique qu'une variable peut être modifiée en arrière-plan par un autre programme (par exemple par une interruption, par un thread, par un autre processus, par le système d'exploitation ou par un autre processeur dans une machine parallèle). Cela nécessite donc de recharger cette variable à chaque fois qu'on y fait référence dans un registre du processeur, et ce *même si elle se trouve déjà dans un de ces registres* (ce qui peut arriver si on a demandé au compilateur d'optimiser le programme).

> **Note :** Le C++ dispose d'un mécanisme plus souple d'isolation des entités propres à un fichier que ce que permet le mot clé `static`, via le mécanisme des espaces de nommages. De plus, le mot clef `static` a une autre signification en C++ dans un autre contexte d'utilisation. L'emploi de ce mot clé pour rendre local à un fichier une variable globale est donc déconseillée en C++, et l'on utilisera de préférence les mécanismes présentés dans le Chapitre 10.

Il existe également des modificateurs pouvant s'appliquer à une variable pour préciser sa constance :

Tableau 3-2. Qualificatifs de constance

Qualificatif	Signification

Qualificatif	Signification
`const`	Ce mot clé est utilisé pour rendre le contenu d'une variable non modifiable. En quelque sorte, la variable devient ainsi une variable en lecture seule. Attention, une telle variable n'est pas forcément une constante : elle peut être modifiée soit par l'intermédiaire d'un autre identificateur, soit par une entité extérieure au programme (comme pour les variables `volatile`). Quand ce mot clé est appliqué à une structure, aucun des champs de la structure n'est accessible en écriture. Bien qu'il puisse paraître étrange de vouloir rendre « constante » une « variable », ce mot clé a une utilité. En particulier, il permet de faire du code plus sûr et dans certains circonstance permet au compilateur d'effectuer des optimisations.
`mutable`	Ce mot clé n'est disponible qu'en C++. Il ne peut être appliqué qu'aux membres des structures. Son rôle est de permettre de passer outre la constance éventuelle d'une structure pour un membre particulier. Ainsi, un champ de structure déclaré `mutable` peut être modifié même si sa structure contenante est déclarée `const`.

Pour déclarer une classe de stockage particulière, il suffit de faire précéder ou suivre le type de la variable par l'un des mots clés `auto`, `static`, `register`, etc. On n'a le droit de n'utiliser que les classes de stockage non contradictoires. Par exemple, `register` et `extern` sont incompatibles, de même que `register` et `volatile`, et `const` et `mutable`. Par contre, `static` et `const`, de même que `const` et `volatile`, peuvent être utilisées simultanément.

Exemple 3-17. Déclaration d'une variable locale statique

```
int appels(void)
{
    static int n = 0;
    return n = n+1;
}
```

Cette fonction mémorise le nombre d'appels qui lui ont été faits dans la variable `n` et renvoie ce nombre. En revanche, la fonction suivante :

```
int appels(void)
{
    int n = 0;
    return n =n + 1;
}
```

renverra toujours `1`. En effet, la variable `n` est créée, initialisée, incrémentée et détruite à chaque appel. Elle ne survit pas à la fin de l'instruction `return`.

Exemple 3-18. Déclaration d'une variable constante

```
const int i=3;
```

`i` prend la valeur `3` et ne peut plus être modifiée.

Les variables globales qui sont définies sans le mot clé `const` sont traitées par le compilateur comme des variables de classe de stockage `extern` par défaut. Ces variables sont donc accessibles à partir de tous les fichiers du programme. En revanche, cette règle n'est pas valide pour les variables définies avec le mot clé `const`. Ces variables sont automatiquement déclarées `static` par le compilateur, ce

qui signifie qu'elles ne sont accessibles que dans le fichier dans lequel elles ont été déclarées. Pour les rendre accessibles aux autres fichiers, il faut impérativement les déclarer avec le mot clé `extern` avant de les définir.

Exemple 3-19. Déclaration de constante externes

```
int i = 12;          /* i est accessible de tous les fichiers. */
const int j = 11;    /* Synonyme de "static const int j = 11;". */

extern const int k;  /* Déclare d'abord la variable k... */
const int k = 12;    /* puis donne la définition. */
```

Notez que toutes les variables définies avec le mot clé `const` doivent être initialisées lors de leur définition. En effet, on ne peut pas modifier la valeur des variables `const`, elles doivent donc avoir une valeur initiale. Enfin, les variables statiques non initialisées prennent la valeur nulle.

Les mots clés `const` et `volatile` demandent au compilateur de réaliser des vérifications additionnelles lors de l'emploi des variables qui ont ces classes de stockage. En effet, le C/C++ assure qu'il est interdit de modifier (du moins sans magouiller) une variable de classe de stockage `const`, et il assure également que toutes les références à une variable de classe de stockage `volatile` se feront sans optimisations dangereuses. Ces vérifications sont basées sur le type des variables manipulées. Dans le cas des types de base, ces vérifications sont simples et de compréhension immédiate. Ainsi, les lignes de code suivantes :

```
const int i=3;
int j=2;

i=j;   /* Illégal : i est de type const int. */
```

génèrent une erreur parce qu'on ne peut pas affecter une valeur de type int à une variable de type const int.

En revanche, pour les types complexes (pointeurs et références en particulier), les mécanismes de vérifications sont plus fins. Nous verrons quels sont les problèmes soulevés par l'emploi des mots clés `const` et `volatile` avec les pointeurs et les références dans le chapitre traitant des pointeurs.

Enfin, en C++ uniquement, le mot clé `mutable` permet de rendre un champ de structure `const` accessible en écriture :

Exemple 3-20. Utilisation du mot clé mutable

```
struct A
{
    int i;           // Non modifiable si A est const.
    mutable int j;   // Toujours modifiable.
};

const A a={1, 1};    // i et j valent 1.

int main(void)
{
    a.i=2;           // ERREUR ! a est de type const A !
    a.j=2;           // Correct : j est mutable.
    return 0;
}
```

Chapitre 4. Les pointeurs et références

Les pointeurs sont des variables très utilisées en C et en C++. Ils doivent être considérés comme des variables, il n'y a rien de sorcier derrière les pointeurs. Cependant, les pointeurs ont un domaine d'application très vaste.

Les références sont des identificateurs synonymes d'autres identificateurs, qui permettent de manipuler certaines notions introduites avec les pointeurs plus souplement. Elles n'existent qu'en C++.

4.1. Notion d'adresse

Tout objet manipulé par l'ordinateur est stocké dans sa mémoire. On peut considérer que cette mémoire est constituée d'une série de « cases », cases dans lesquelles sont stockées les valeurs des variables ou les instructions du programme. Pour pouvoir accéder à un objet (la valeur d'une variable ou les instructions à exécuter par exemple), c'est-à-dire au contenu de la case mémoire dans laquelle cet objet est enregistré, il faut connaître le numéro de cette case. Autrement dit, il faut connaître l'emplacement en mémoire de l'objet à manipuler. Cet emplacement est appelé l'*adresse* de la case mémoire, et par extension, l'*adresse de la variable* ou l'*adresse de la fonction* stockée dans cette case et celles qui la suivent.

Toute case mémoire a une adresse unique. Lorsqu'on utilise une variable ou une fonction, le compilateur manipule l'adresse de cette dernière pour y accéder. C'est lui qui connaît cette adresse, le programmeur n'a pas à s'en soucier.

4.2. Notion de pointeur

Une adresse est une valeur. On peut donc stocker cette valeur dans une variable. Les *pointeurs* sont justement des variables qui contiennent l'adresse d'autres objets, par exemple l'adresse d'une autre variable. On dit que le pointeur *pointe* sur la variable *pointée*. Ici, *pointer* signifie « faire référence à ». Les adresses sont généralement des valeurs constantes, car en général un objet ne se déplace pas en mémoire. Toutefois, la valeur d'un pointeur peut changer. Cela ne signifie pas que la variable pointée est déplacée en mémoire, mais plutôt que le pointeur pointe sur autre chose.

Afin de savoir ce qui est pointé par un pointeur, les pointeurs disposent d'un type. Ce type est construit à partir du type de l'objet pointé. Cela permet au compilateur de vérifier que les manipulations réalisées en mémoire par l'intermédiaire du pointeur sont valides. Le type des pointeur se lit « pointeur de ... », où les points de suspension représentent le nom du type de l'objet pointé.

Les pointeurs se déclarent en donnant le type de l'objet qu'ils devront pointer, suivi de leur identificateur précédé d'une étoile :

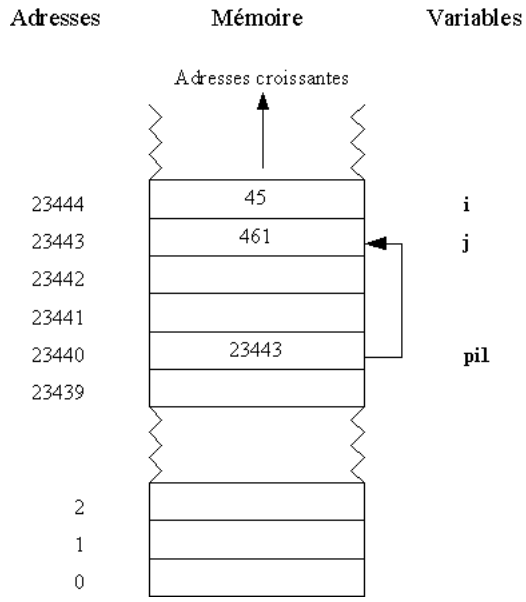
```
int *pi;   // pi est un pointeur d'entier.
```

Note : Si plusieurs pointeurs doivent être déclarés, l'étoile doit être répétée :

```
int *pi1, *pi2, j, *pi3;
```

Ici, `pi1`, `pi2` et `pi3` sont des pointeurs d'entiers et `j` est un entier.

Figure 4-1. Notion de pointeur et d'adresse



Il est possible de faire un pointeur sur une structure dans une structure en indiquant le nom de la structure comme type du pointeur :

```
typedef struct nom
{
    struct nom *pointeur;   /* Pointeur sur une structure "nom". */
    ...
} MaStructure;
```

Ce type de construction permet de créer des listes de structures, dans lesquelles chaque structure contient l'adresse de la structure suivante dans la liste. Nous verrons plus loin un exemple d'utilisation de ce genre de structure.

Il est également possible de créer des pointeurs sur des fonctions, et d'utiliser ces pointeurs pour paramétrer un algorithme, dont le comportement dépendra des fonctions ainsi pointées. Nous détaillerons plus loin ce type d'utilisation des pointeurs.

4.3. Déréférencement, indirection

Un pointeur ne servirait strictement à rien s'il n'y avait pas de possibilité d'accéder à l'adresse d'une variable ou d'une fonction (on ne pourrait alors pas l'initialiser) ou s'il n'y avait pas moyen d'accéder à l'objet référencé par le pointeur (la variable pointée ne pourrait pas être manipulée ou la fonction pointée ne pourrait pas être appelée).

Ces deux opérations sont respectivement appelées *indirection* et *déréférencement*. Il existe deux opérateurs permettant de récupérer l'adresse d'un objet et d'accéder à l'objet pointé. Ces opérateurs sont respectivement `&` et `*`.

Il est très important de s'assurer que les pointeurs que l'on manipule sont tous initialisés (c'est-à-dire contiennent l'adresse d'un objet valide, et pas n'importe quoi). En effet, accéder à un pointeur

non initialisé revient à lire ou, plus grave encore, à écrire dans la mémoire à un endroit complètement aléatoire (selon la valeur initiale du pointeur lors de sa création). En général, on initialise les pointeurs dès leur création, ou, s'ils doivent être utilisés ultérieurement, on les initialise avec le pointeur nul. Cela permettra de faire ultérieurement des tests sur la validité du pointeur ou au moins de détecter les erreurs. En effet, l'utilisation d'un pointeur initialisé avec le pointeur nul génère souvent une faute de protection du programme, que tout bon débogueur est capable de détecter. Le pointeur nul se note `NULL`.

> **Note :** `NULL` est une macro définie dans le fichier d'en-tête `stdlib.h`. En C, elle représente la valeur d'une adresse invalide. Malheureusement, cette valeur peut ne pas être égale à l'adresse `0` (certains compilateurs utilisent la valeur `-1` pour `NULL` par exemple). C'est pour cela que cette macro a été définie, afin de représenter, selon le compilateur, la bonne valeur. Voir le Chapitre 5 pour plus de détails sur les macros et sur les fichiers d'en-tête.
>
> La norme du C++ fixe la valeur nulle des pointeurs à `0`. Par conséquent, les compilateurs C/C++ qui définissent `NULL` comme étant égal à `-1` posent un problème de portabilité certain, puisque un programme C qui utilise `NULL` n'est plus valide en C++. Par ailleurs, un morceau de programme C++ compilable en C qui utiliserait la valeur `0` ne serait pas correct en C.
>
> Il faut donc faire un choix : soit utiliser `NULL` en C et `0` en C++, soit utiliser `NULL` partout, quitte à redéfinir la macro `NULL` pour les programmes C++ (solution qui me semble plus pratique).

Exemple 4-1. Déclaration de pointeurs

```
int i=0;       /* Déclare une variable entière. */
int *pi;       /* Déclare un pointeur sur un entier. */
pi=&i;         /* Initialise le pointeur avec l'adresse de cette
                  variable. */
*pi = *pi+1;   /* Effectue un calcul sur la variable pointée par pi,
                  c'est-à-dire sur i lui-même, puisque pi contient
                  l'adresse de i. */

               /* À ce stade, i ne vaut plus 0, mais 1. */
```

Il est à présent facile de comprendre pourquoi il faut répéter l'étoile dans la déclaration de plusieurs pointeurs :

```
int *p1, *p2, *p3;
```

signifie syntaxiquement : `p1`, `p2` et `p3` sont des pointeurs d'entiers, mais aussi `*p1`, `*p2` et `*p3` sont des entiers.

Si l'on avait écrit :

```
int *p1, p2, p3;
```

seul `p1` serait un pointeur d'entier. `p2` et `p3` seraient des entiers.

L'accès aux champs d'une structure par le pointeur sur cette structure se fera avec l'opérateur '`->`', qui remplace '`(*).`'.

Exemple 4-2. Utilisation de pointeurs de structures

```
struct Client
{
    int Age;
```

```
};

Client structure1;
Client *pstr = &structure1;
pstr->Age = 35;   /* On aurait pu écrire (*pstr).Age=35; */
```

4.4. Notion de référence

En plus des pointeurs, le C++ permet de créer des références. Les *références* sont des synonymes d'identificateurs. Elles permettent de manipuler une variable sous un autre nom que celui sous laquelle cette dernière a été déclarée.

Note : Les références n'existent qu'en C++. Le C ne permet pas de créer des références.

Par exemple, si « `id` » est le nom d'une variable, il est possible de créer une référence « `ref` » de cette variable. Les deux identificateurs `id` et `ref` représentent alors la même variable, et celle-ci peut être accédée et modifiée à l'aide de ces deux identificateurs indistinctement.

Toute référence doit se référer à un identificateur : il est donc impossible de déclarer une référence sans l'initialiser. De plus, la déclaration d'une référence ne crée pas un nouvel objet comme c'est le cas pour la déclaration d'une variable par exemple. En effet, les références se rapportent à des identificateurs déjà existants.

La syntaxe de la déclaration d'une référence est la suivante :

```
type &référence = identificateur;
```

Après cette déclaration, référence peut être utilisé partout où identificateur peut l'être. Ce sont des synonymes.

Exemple 4-3. Déclaration de références

```
int i=0;
int &ri=i;   // Référence sur la variable i.
ri=ri+i;     // Double la valeur de i (et de ri).
```

Il est possible de faire des références sur des valeurs numériques. Dans ce cas, les références doivent être déclarées comme étant constantes, puisqu'une valeur est une constante :

```
const int &ri=3;  // Référence sur 3.
int &error=4;     // Erreur ! La référence n'est pas constante.
```

4.5. Lien entre les pointeurs et les références

Les références et les pointeurs sont étroitement liés. En effet, une variable et ses différentes références ont la même adresse, puisqu'elles permettent d'accéder à un même objet. Utiliser une référence pour

manipuler un objet revient donc exactement au même que de manipuler un pointeur constant contenant l'adresse de cet objet. Les références permettent simplement d'obtenir le même résultat que les pointeurs, mais avec une plus grande facilité d'écriture.

Cette similitude entre les pointeurs et les références se retrouve au niveau syntaxique. Par exemple, considérons le morceau de code suivant :

```
int i=0;
int *pi=&i;
*pi=*pi+1;   // Manipulation de i via pi.
```

et faisons passer l'opérateur & de la deuxième ligne à gauche de l'opérateur d'affectation :

```
int i=0;
int &*pi=i;   // Cela génère une erreur de syntaxe mais nous
              // l'ignorons pour les besoins de l'explication.
*pi=*pi+1;
```

Maintenant, comparons avec le morceau de code équivalent suivant :

```
int i=0;
int &ri=i;
ri=ri+1;      // Manipulation de i via ri.
```

Nous constatons que la référence `ri` peut être identifiée avec l'expression `*pi`, qui représente bel et bien la variable `i`. Ainsi, la référence `ri` encapsule la manipulation de l'adresse de la variable `i` et s'utilise comme l'expression `*pi`. Cela permet de comprendre l'origine de la syntaxe de déclaration des références. La différence se trouve ici dans le fait que les références doivent être initialisées d'une part, et que l'on n'a pas à effectuer le déréférencement d'autre part. Les références sont donc beaucoup plus faciles à manipuler que les pointeurs, et permettent de faire du code beaucoup plus sûr.

4.6. Passage de paramètres par variable ou par valeur

Il y a deux méthodes pour passer des variables en paramètre dans une fonction : le *passage par valeur* et le *passage par variable*. Ces méthodes sont décrites ci-dessous.

4.6.1. Passage par valeur

La valeur de l'expression passée en paramètre est copiée dans une variable locale. C'est cette variable qui est utilisée pour faire les calculs dans la fonction appelée.

Si l'expression passée en paramètre est une variable, son contenu est copié dans la variable locale. Aucune modification de la variable locale dans la fonction appelée ne modifie la variable passée en paramètre, parce que ces modifications ne s'appliquent qu'à une copie de cette dernière.

Le C ne permet de faire que des passages par valeur.

Exemple 4-4. Passage de paramètre par valeur

```
#include <stdlib.h>

void test(int j)    /* j est la copie de la valeur passée en
                       paramètre */
{
    j=3;            /* Modifie j, mais pas la variable fournie
                       par l'appelant. */
    return;
}

int main(void)
{
    int i=2;
    test(i);        /* Le contenu de i est copié dans j.
                       i n'est pas modifié. Il vaut toujours 2. */
    test(2);        /* La valeur 2 est copiée dans j. */
    return EXIT_SUCCESS;
}
```

4.6.2. Passage par variable

La deuxième technique consiste à passer non plus la valeur des variables comme paramètre, mais à passer les variables elles-mêmes. Il n'y a donc plus de copie, plus de variable locale. Toute modification du paramètre dans la fonction appelée entraîne la modification de la variable passée en paramètre.

Le C ne permet pas de faire ce type de passage de paramètres (le C++ le permet en revanche).

Exemple 4-5. Passage de paramètre par variable en Pascal

```
Var i : integer;

Procedure test(Var j : integer)
Begin
            {La variable j est strictement égale
             à la variable passée en paramètre.}
   j:=2;    {Ici, cette variable est modifiée.}
End;

Begin
   i:=3;    {Initialise i à 3}
   test(i); {Appelle la fonction. La variable i est passée en
             paramètres, pas sa valeur. Elle est modifiée par
             la fonction test.}

     {Ici, i vaut 2.}
End.
```

Puisque la fonction attend une variable en paramètre, on ne peut plus appeler `test` avec une valeur (`test(3)` est maintenant interdit, car `3` n'est pas une variable : on ne peut pas le modifier).

4.6.3. Avantages et inconvénients des deux méthodes

Les passages par variables sont plus rapides et plus économes en mémoire que les passages par valeur, puisque les étapes de la création de la variable locale et la copie de la valeur ne sont pas faites. Il faut donc éviter les passages par valeur dans les cas d'appels récursifs de fonction ou de fonctions travaillant avec des grandes structures de données (matrices par exemple).

Les passages par valeurs permettent d'éviter de détruire par mégarde les variables passées en paramètre. Si l'on veut se prévenir de la destruction accidentelle des paramètres passés par variable, il faut utiliser le mot clé `const`. Le compilateur interdira alors toute modification de la variable dans la fonction appelée, ce qui peut parfois obliger cette fonction à réaliser des copies de travail en local.

4.6.4. Comment passer les paramètres par variable en C ?

Il n'y a qu'une solution : passer l'adresse de la variable. Cela constitue donc une application des pointeurs.

Voici comment l'Exemple 4-5 serait programmé en C :

Exemple 4-6. Passage de paramètre par variable en C

```
#include <stdlib.h>

void test(int *pj)  /* test attend l'adresse d'un entier... */
{
    *pj=2;          /* ... pour le modifier. */
    return;
}

int main(void)
{
    int i=3;
    test(&i);       /* On passe l'adresse de i en paramètre. */
    /* Ici, i vaut 2. */
    return EXIT_SUCCESS;
}
```

À présent, il est facile de comprendre la signification de & dans l'appel de `scanf` : les variables à entrer sont passées par variable.

4.6.5. Passage de paramètres par référence

La solution du C est exactement la même que celle du Pascal du point de vue sémantique. En fait, le Pascal procède exactement de la même manière en interne, mais la manipulation des pointeurs est masquée par le langage. Cependant, plusieurs problèmes se posent au niveau syntaxique :

- la syntaxe est lourde dans la fonction, à cause de l'emploi de l'opérateur `*` devant les paramètres ;
- la syntaxe est dangereuse lors de l'appel de la fonction, puisqu'il faut systématiquement penser à utiliser l'opérateur `&` devant les paramètres. Un oubli devant une variable de type entier et la valeur de l'entier est utilisée à la place de son adresse dans la fonction appelée (plantage assuré, essayez avec `scanf`).

Le C++ permet de résoudre tous ces problèmes à l'aide des références. Au lieu de passer les adresses des variables, il suffit de passer les variables elles-mêmes en utilisant des paramètres sous la forme de références. La syntaxe des paramètres devient alors :

```
type &identificateur [, type &identificateur [...]]
```

Exemple 4-7. Passage de paramètre par référence en C++

```
#include <stdlib.h>

void test(int &i)   // i est une référence du paramètre constant.
{
    i = 2;          // Modifie le paramètre passé en référence.
    return;
}

int main(void)
{
    int i=3;
    test(i);
    // Après l'appel de test, i vaut 2.
    // L'opérateur & n'est pas nécessaire pour appeler
    // test.
    return EXIT_SUCCESS;
}
```

Il est recommandé, pour des raisons de performances, de passer par référence tous les paramètres dont la copie peut prendre beaucoup de temps (en pratique, seuls les types de base du langage pourront être passés par valeur). Bien entendu, il faut utiliser des références constantes au maximum afin d'éviter les modifications accidentelles des variables de la fonction appelante dans la fonction appelée. En revanche, les paramètres de retour des fonctions ne devront pas être déclarés comme des références constantes, car on ne pourrait pas les écrire si c'était le cas.

Exemple 4-8. Passage de paramètres constant par référence

```
typedef struct
{
    ...
} structure;

void ma_fonction(const structure & s)
{
    ...
    return ;
}
```

Dans cet exemple, `s` est une référence sur une structure constante. Le code se trouvant à l'intérieur de la fonction ne peut donc pas utiliser la référence `s` pour modifier la structure (on notera cependant que c'est la fonction elle-même qui s'interdit l'écriture dans la variable `s`. `const` est donc un mot clé « coopératif ». Il n'est pas possible à un programmeur d'empêcher ses collègues d'écrire dans ses variables avec le mot clé `const`. Nous verrons dans le Chapitre 7 que le C++ permet de pallier ce problème grâce à une technique appelée l'encapsulation.).

Un autre avantage des références constantes pour les passages par variables est que si le paramètre n'est pas une variable ou, s'il n'est pas du bon type, une variable locale du type du paramètre est créée et initialisée avec la valeur du paramètre transtypé.

Exemple 4-9. Création d'un objet temporaire lors d'un passage par référence

```
#include <stdlib.h>

void test(const int &i)
{
    ...       // Utilisation de la variable i
              // dans la fonction test. La variable
              // i est créée si nécessaire.
    return ;
}

int main(void)
{
    test(3);   // Appel de test avec une constante.
    return EXIT_SUCCESS;
}
```

Au cours de cet appel, une variable locale est créée (la variable `i` de la fonction `test`), et `3` lui est affectée.

4.7. Références et pointeurs constants et volatiles

L'utilisation des mots clés `const` et `volatile` avec les pointeurs et les références est un peu plus compliquée qu'avec les types simples. En effet, il est possible de déclarer des pointeurs sur des variables, des pointeurs constants sur des variables, des pointeurs sur des variables constantes et des pointeurs constants sur des variables constantes (bien entendu, il en est de même avec les références). La position des mots clés `const` et `volatile` dans les déclarations des types complexes est donc extrêmement importante. En général, les mots clés `const` et `volatile` caractérisent ce qui les précède dans la déclaration, si l'on adopte comme règle de toujours les placer après les types de base. Par exemple, l'expression suivante :

```
const int *pi;
```

peut être réécrite de la manière suivante :

```
int const *pi;
```

puisque le mot clé `const` est interchangeable avec le type le plus simple dans une déclaration. Ce mot clé caractérise donc le type int, et `pi` est un pointeur sur un entier constant. En revanche, dans l'exemple suivant :

```
int j;
int * const pi=&j;
```

`pi` est déclaré comme étant constant, et de type pointeur d'entier. Il s'agit donc d'un pointeur constant sur un entier non constant, que l'on initialise pour référencer la variable `j`.

Note : Les déclarations C++ peuvent devenir très compliquées et difficiles à lire. Il existe une astuce qui permet de les interpréter facilement. Lors de l'analyse de la déclaration d'un identificateur `X`, il faut toujours commencer par une phrase du type « `X est un ...` ». Pour trouver la suite de la phrase, il suffit de lire la déclaration en partant de l'identificateur et de suivre l'ordre imposé par les priorités des opérateurs. Cet ordre peut être modifié par la présence de parenthèses. Vous trouverez en annexe les priorités de tous les opérateurs du C++.

Ainsi, dans l'exemple suivant :

```
const int *pi[12];
void (*pf)(int * const pi);
```

la première déclaration se lit de la manière suivante : « pi (`pi`) est un tableau (`[]`) de 12 (`12`) pointeurs (`*`) d'entiers (`int`) constants (`const`) ». La deuxième déclaration se lit : « pf (`pf`) est un pointeur (`*`) de fonction (`()`) de pi (`pi`), qui est lui-même une constante (`const`) de type pointeur (`*`) d'entier (`int`). Cette fonction ne renvoie rien (`void`) ».

Le C et le C++ n'autorisent que les écritures qui conservent ou augmentent les propriétés de constance et de volatilité. Par exemple, le code suivant est correct :

```
char *pc;
const char *cpc;

cpc=pc;   /* Le passage de pc à cpc augmente la constance. */
```

parce qu'elle signifie que si l'on peut écrire dans une variable par l'intermédiaire du pointeur `pc`, on peut s'interdire de le faire en utilisant `cpc` à la place de `pc`. En revanche, si l'on n'a pas le droit d'écrire dans une variable, on ne peut en aucun cas se le donner.

Cependant, les règles du langage relatives à la modification des variables peuvent parfois paraître étranges. Par exemple, le langage interdit une écriture telle que celle-ci :

```
char *pc;
const char **ppc;

ppc = &pc;   /* Interdit ! */
```

Pourtant, cet exemple ressemble beaucoup à l'exemple précédent. On pourrait penser que le fait d'affecter un pointeur de pointeur de variable à un pointeur de pointeur de variable constante revient à s'interdire d'écrire dans une variable qu'on a le droit de modifier. Mais en réalité, cette écriture va contre les règles de constances, parce qu'elle permettrait de modifier une variable constante. Pour s'en convaincre, il faut regarder l'exemple suivant :

```
const char c='a';      /* La variable constante. */
char *pc;              /* Pointeur par l'intermédiaire duquel
                          nous allons modifier c. */
const char **ppc=&pc;  /* Interdit, mais supposons que ce ne le
                          soit pas. */
*ppc=&c;               /* Parfaitement légal. */
*pc='b';               /* Modifie la variable c. */
```

Que s'est-il passé ? Nous avons, par l'intermédiaire de `ppc`, affecté l'adresse de la constante `c` au pointeur `pc`. Malheureusement, `pc` n'est pas un pointeur de constante, et cela nous a permis de modifier la constante `c`.

Afin de gérer correctement cette situation (et les situations plus complexes qui utilisent des triples pointeurs ou encore plus d'indirection), le C et le C++ interdisent l'affectation de tout pointeur dont les propriétés de constance et de volatilité sont moindres que celles du pointeur cible. La règle exacte est la suivante :

1. On note `cv` les différentes qualifications de constance et de volatilité possibles (à savoir : `const volatile`, `const`, `volatile` ou aucune classe de stockage).
2. Si le pointeur source est un pointeur `cvs,0` de pointeur `cvs,1` de pointeur ... de pointeur `cvs,n-1` de type `Ts cvs,n`, et que le pointeur destination est un pointeur `cvd,0` de pointeur `cvd,1` de pointeur ... de pointeur `cvd,n-1` de type `Td cvs,n`, alors l'affectation de la source à la destination n'est légale que si :
   - les types source `Ts` et destination `Td` sont compatibles ;
   - il existe un nombre entier strictement positif `N` tel que, quel que soit `j` supérieur ou égal à `N`, on ait :
     - si `const` apparaît dans `cvs,j`, alors `const` apparaît dans `cvd,j` ;
     - si `volatile` apparaît dans `cvs,j`, alors `volatile` apparaît dans `cvd,j` ;
     - et tel que, quel que soit `0<k<N`, `const` apparaisse dans `cvd,k`.

Ces règles sont suffisamment compliquées pour ne pas être apprises. Les compilateurs se chargeront de signaler les erreurs s'il y en a en pratique. Par exemple :

```
const char c='a';
const char *pc;
const char **ppc=&pc;  /* Légal à présent. */
*ppc=&c;
*pc='b';               /* Illégal (pc a changé de type). */
```

L'affectation de double pointeur est à présent légale, parce que le pointeur source a changé de type (on ne peut cependant toujours pas modifier le caractère `c`).

Il existe une exception notable à ces règles : l'initialisation des chaînes de caractères. Les chaînes de caractères telles que :

```
"Bonjour tout le monde !"
```

sont des chaînes de caractères constantes. Par conséquent, on ne peut théoriquement affecter leur adresse qu'à des pointeurs de caractères constants :

```
const char *pc="Coucou !"; /* Code correct. */
```

Cependant, il a toujours été d'usage de réaliser l'initialisation des chaînes de caractères de la manière suivante :

```
char *pc="Coucou !";        /* Théoriquement illégal, mais toléré
                               par compatibilité avec le C. */
```

Par compatibilité, le langage fournit donc une conversion implicite entre « `const char *` » et « `char *` ». Cette facilité ne doit pas pour autant vous inciter à transgresser les règles de constance : utilisez les pointeurs sur les chaînes de caractères constants autant que vous le pourrez (quitte à réaliser quelques copies de chaînes lorsqu'un pointeur de caractère simple doit être utilisé). Sur certains systèmes, l'écriture dans une chaîne de caractères constante peut provoquer un plantage immédiat du programme.

4.8. Arithmétique des pointeurs

Il est possible d'effectuer des opérations arithmétiques sur les pointeurs.

Les seules opérations valides sont les opérations externes (addition et soustraction des entiers) et la soustraction de pointeurs. Elles sont définies comme suit (la soustraction d'un entier est considérée comme l'addition d'un entier négatif) :

```
p + i = adresse contenue dans p + i*taille(élément pointé par p)
```

et :

```
p2 - p1 = (adresse contenue dans p2 - adresse contenue dans p1) /
          taille(éléments pointés par p1 et p2)
```

Si `p` est un pointeur d'entier, `p+1` est donc le pointeur sur l'entier qui suit immédiatement celui pointé par `p`. On retiendra surtout que l'entier qu'on additionne au pointeur est multiplié par la taille de l'élément pointé pour obtenir la nouvelle adresse.

Le type du résultat de la soustraction de deux pointeurs est très dépendant de la machine cible et du modèle mémoire du programme. En général, on ne pourra jamais supposer que la soustraction de deux pointeurs est un entier (que les chevronnés du C me pardonnent, mais c'est une erreur *très grave*). En effet, ce type peut être insuffisant pour stocker des adresses (une machine peut avoir des adresses sur 64 bits et des données sur 32 bits). Pour résoudre ce problème, le fichier d'en-tête `stdlib.h` contient la définition du type à utiliser pour la différence de deux pointeurs. Ce type est nommé ptrdiff_t.

Exemple 4-10. Arithmétique des pointeurs

```
int i, j;
ptrdiff_t delta = &i - &j;  /* Correct */
int error = &i - &j;        /* Peut marcher, mais par chance. */
```

Il est possible de connaître la taille d'un élément en caractères en utilisant l'opérateur `sizeof`. Il a la syntaxe d'une fonction :

```
sizeof(type|expression)
```

Il attend soit un type, soit une expression. La valeur retournée est soit la taille du type en caractères, soit celle du type de l'expression. Dans le cas des tableaux, il renvoie la taille totale du tableau. Si son

argument est une expression, celle-ci n'est pas évaluée (donc si il contient un appel à une fonction, celle-ci n'est pas appelée). Par exemple :

```
sizeof(int)
```

renvoie la taille d'un entier en caractères, et :

```
sizeof(2+3)
```

renvoie la même taille, car `2+3` est de type entier. `2+3` n'est pas calculé.

> **Note :** L'opérateur `sizeof` renvoie la taille des types en tenant compte de leur alignement. Cela signifie par exemple que même si un compilateur espace les éléments d'un tableau afin de les aligner sur des mots mémoire de la machine, la taille des éléments du tableau sera celle des objets de même type qui ne se trouvent pas dans ce tableau (ils devront donc être alignés eux aussi). On a donc toujours l'égalité suivante :
>
> ```
> sizeof(tableau) = sizeof(élément) * nombre d'éléments
> ```

4.9. Utilisation des pointeurs avec les tableaux

Les tableaux sont étroitement liés aux pointeurs parce que, de manière interne, l'accès aux éléments des tableaux se fait par manipulation de leur adresse de base, de la taille des éléments et de leurs indices. En fait, l'adresse du n-ième élément d'un tableau est calculée avec la formule :

```
Adresse_n = Adresse_Base + n*taille(élément)
```

où `taille(élément)` représente la taille de chaque élément du tableau et `Adresse_Base` l'adresse de base du tableau. Cette adresse de base est l'adresse du début du tableau, c'est donc à la fois l'adresse du tableau et l'adresse de son premier élément.

Ce lien apparaît au niveau du langage dans les conversions implicites de tableaux en pointeurs, et dans le passage des tableaux en paramètre des fonctions.

Afin de pouvoir utiliser l'arithmétique des pointeurs pour manipuler les éléments des tableaux, le C++ effectue les conversions implicites suivantes lorsque nécessaire :

- tableau vers pointeur d'élément ;
- pointeur d'élément vers tableau.

Cela permet de considérer les expressions suivantes comme équivalentes :

```
identificateur[n]
```

et :

```
*(identificateur + n)
```

si `identificateur` est soit un identificateur de tableau, soit celui d'un pointeur.

Exemple 4-11. Accès aux éléments d'un tableau par pointeurs

```
int tableau[100];
int *pi=tableau;

tableau[3]=5;   /* Le 4ème élément est initialisé à 5 */
*(tableau+2)=4; /* Le 3ème élément est initialisé à 4 */
pi[5]=1;        /* Le 6ème élément est initialisé à 1 */
```

Note : Le langage C++ impose que l'adresse suivant le dernier élément d'un tableau doit toujours être valide. Cela ne signifie absolument pas que la zone mémoire référencée par cette adresse est valide, bien au contraire, mais plutôt que cette adresse est valide. Il est donc garantit que cette adresse ne sera pas le pointeur NULL par exemple, ni toute autre valeur spéciale qu'un pointeur ne peut pas stocker. Il sera donc possible de faire des calculs d'arithmétique des pointeurs avec cette adresse, même si elle ne devra jamais être déréférencée, sous peine de voir le programme planter.

On prendra garde à certaines subtilités. Les conversions implicites sont une facilité introduite par le compilateur, mais en réalité, les tableaux ne sont pas des pointeurs, ce sont des variables comme les autres, à ceci près : leur type est convertible en pointeur sur le type de leurs éléments. Il en résulte parfois quelques ambiguïtés lorsqu'on manipule les adresses des tableaux. En particulier, on a l'égalité suivante :

```
&tableau == tableau
```

en raison du fait que l'adresse du tableau est la même que celle de son premier élément. Il faut bien comprendre que dans cette expression, une conversion a lieu. Cette égalité n'est donc pas exacte en théorie. En effet, si c'était le cas, on pourrait écrire :

```
*&tableau == tableau
```

puisque les opérateurs `*` et `&` sont conjugués, d'où :

```
tableau == *&tableau = *(&tableau) == *(tableau) == t[0]
```

ce qui est faux (le type du premier élément n'est en général pas convertible en type pointeur.).

La conséquence la plus importante de la conversion tableau vers pointeur se trouve dans le passage par variable des tableaux dans une fonction. Lors du passage d'un tableau en paramètre d'une fonction, la conversion implicite a lieu, les tableaux sont donc toujours passés par variable, jamais par valeur. Il est donc faux d'utiliser des pointeurs pour les passer en paramètre, car le paramètre aurait le type pointeur de tableau. On ne modifierait pas le tableau, mais bel et bien le pointeur du tableau. Le programme aurait donc de fortes chances de planter. Si un passage par valeur du tableau est désiré, il faut l'encapsuler dans une structure.

4.10. Les chaînes de caractères : pointeurs et tableaux à la fois !

On a vu dans le premier chapitre que les chaînes de caractères n'existaient pas en C/C++. Ce sont en réalité des tableaux de caractères dont le dernier caractère est le caractère nul.

Cela a plusieurs conséquences. La première, c'est que les chaînes de caractères sont aussi des pointeurs sur des caractères, ce qui se traduit dans la syntaxe de la déclaration d'une chaîne de caractères constante :

```
const char *identificateur = "chaîne";
```

`identificateur` est déclaré ici comme étant un pointeur de caractère, puis il est initialisé avec l'adresse de la chaîne de caractères constante `"chaîne"`.

La deuxième est le fait qu'on ne peut pas faire, comme en Pascal, des affectations de chaînes de caractères, ni des comparaisons. Par exemple, si « `nom1` » et « `nom2` » sont des chaînes de caractères, l'opération :

```
nom1=nom2;
```

n'est pas l'affectation du contenu de `nom2` à `nom1`. C'est une affectation de pointeur : le pointeur `nom1` est égal au pointeur `nom2` et pointent sur *la même chaîne* ! Une modification de la chaîne pointée par `nom1` entraîne donc la modification de la chaîne pointée par `nom2`...

De même, le test `nom1==nom2` est un test entre pointeurs, pas entre chaînes de caractères. Même si deux chaînes sont égales, le test sera faux si elles ne sont pas au même emplacement mémoire.

Il existe dans la bibliothèque C de nombreuses fonctions permettant de manipuler les chaînes de caractères. Par exemple, la copie d'une chaîne de caractères dans une autre se fera avec les fonctions `strcpy` et `strncpy`, la comparaison de deux chaînes de caractères pourra être réalisée à l'aide des fonctions `strcmp` et `strncmp`, et la détermination de la longueur d'une chaîne de caractères à l'aide de la fonction `strlen`. Je vous invite à consulter la documentation de votre environnement de développement ou la bibliographie pour découvrir toutes les fonctions de manipulation des chaînes de caractères. Nous en verrons un exemple d'utilisation dans la section suivante.

4.11. Allocation dynamique de mémoire

Les pointeurs sont surtout utilisés pour créer un nombre quelconque de variables, ou des variables de taille quelconque, en cours d'exécution du programme.

En temps normal, les variables sont créées automatiquement lors de leur définition. Cela est faisable parce que les variables à créer ainsi que leurs tailles sont connues au moment de la compilation (c'est le but des déclarations que d'indiquer la structure et la taille des objets, et plus généralement de donner les informations nécessaires à leur utilisation). Par exemple, une ligne comme :

```
int tableau[10000];
```

signale au compilateur qu'une variable `tableau` de 10000 entiers doit être créée. Le programme s'en chargera donc automatiquement lors de l'exécution.

Mais supposons que le programme gère une liste de personnes. On ne peut pas savoir à l'avance combien de personnes seront entrées, le compilateur ne peut donc pas faire la réservation de l'espace mémoire automatiquement. C'est au programmeur de le faire. Cette réservation de mémoire (appelée encore *allocation*) doit être faite pendant l'exécution du programme. La différence avec la déclaration de tableau précédente, c'est que le nombre de personnes et donc la quantité de mémoire à allouer, est variable. Il faut donc faire ce qu'on appelle une *allocation dynamique de mémoire*.

4.11.1. Allocation dynamique de mémoire en C

Il existe deux principales fonctions C permettant de demander de la mémoire au système d'exploitation et de la lui restituer. Elles utilisent toutes les deux les pointeurs, parce qu'une variable allouée dynamiquement n'a pas d'identificateur étant donné qu'elle n'était a priori pas connue à la compilation, et n'a donc pas pu être déclarée. Les pointeurs utilisés par ces fonctions C n'ont pas de type. On les référence donc avec des pointeurs non typés. Leur syntaxe est la suivante :

```
malloc(taille)
free(pointeur)
```

`malloc` (abréviation de « Memory ALLOCation ») alloue de la mémoire. Elle attend comme paramètre la taille de la zone de mémoire à allouer et renvoie un pointeur non typé (void *).

`free` (pour « FREE memory ») libère la mémoire allouée. Elle attend comme paramètre le pointeur sur la zone à libérer et ne renvoie rien.

Lorsqu'on alloue une variable typée, on doit faire un transtypage du pointeur renvoyé par `malloc` en pointeur de ce type de variable.

Pour utiliser les fonctions `malloc` et `free`, vous devez mettre au début de votre programme la ligne :

```
#include <stdlib.h>
```

Son rôle est similaire à celui de la ligne `#include <stdio.h>`. Vous verrez sa signification dans le chapitre concernant le préprocesseur.

L'exemple suivant va vous présenter un programme C classique qui manipule des pointeurs. Ce programme réalise des allocations dynamiques de mémoire et manipule une liste de structures dynamiquement, en fonction des entrées que fait l'utilisateur. Les techniques de saisies de paramètres présentées dans le premier chapitre sont également revues. Ce programme vous présente aussi comment passer des paramètres par variable, soit pour optimiser le programme, soit pour les modifier au sein des fonctions appelées. Enfin, l'utilisation du mot clef `const` avec les pointeurs est également illustrée.

Exemple 4-12. Allocation dynamique de mémoire en C

```
#include <stdio.h>
#include <stdlib.h>  /* Fichier d'en-tête pour malloc et free. */
#include <string.h>  /* Fichier d'en-tête pour strcpy, strlen et de strcmp. */

/* Type de base d'un élément de liste de personne. */
typedef struct person
{
    char *name;           /* Nom de la personne. */
    char *address;        /* Adresse de la personne. */
    struct person *next;  /* Pointeur sur l'élément suivant. */
} Person;

typedef Person *People;   /* Type de liste de personnes. */

/* Fonctions de gestion des listes de personnes : */

/* Fonction d'initialisation d'une liste de personne.
   La liste est passée par variable pour permettre son initialisation. */
```

```
void init_list(People *lst)
{
   *lst = NULL;
}

/* Fonction d'ajout d'une personne. Les paramètres de la personne
   sont passés par variables, mais ne peuvent être modifiés car
   ils sont constants. Ce sont des chaînes de caractères C, qui
   sont donc assimilées à des pointeurs de caractères constants. */
int add_person(People *lst, const char *name, const char *address)
{
    /* Crée un nouvel élément : */
    Person *p = (Person *) malloc(sizeof(Person));
    if (p != NULL)
    {
        /* Alloue la mémoire pour le nom et l'adresse. Attention,
           il faut compter le caractère nul terminal des chaînes : */
        p->name = (char *) malloc((strlen(name) + 1) * sizeof(char));
        p->address = (char *) malloc((strlen(address) + 1) * sizeof(char));
        if (p->name != NULL && p->address != NULL)
        {
            /* Copie le nom et l'adresse : */
            strcpy(p->name, name);
            strcpy(p->address, address);
            p->next = *lst;
            *lst = p;
        }
        else
        {
            free(p);
            p = NULL;
        }
    }
    return (p != NULL);
}

/* Fonction de suppression d'une personne.
   La structure de la liste est modifiée par la suppression
   de l'élément de cette personne. Cela peut impliquer la modification
   du chaînage de l'élément précédent, ou la modification de la tête
   de liste elle-même. */
int remove_person(People *lst, const char *name)
{
    /* Recherche la personne et son antécédant : */
    Person *prev = NULL;
    Person *p = *lst;
    while (p != NULL)
    {
        /* On sort si l'élément courant est la personne recherchée : */
        if (strcmp(p->name, name) == 0)
            break;
        /* On passe à l'élément suivant sinon : */
        prev = p;
        p = p->next;
    }
    if (p != NULL)
    {
```

```
            /* La personne a été trouvée, on la supprime de la liste : */
            if (prev == NULL)
            {
                /* La personne est en tête de liste, on met à jour
                   le pointeur de tête de liste : */
                *lst = p->next;
            }
            else
            {
                /* On met à jour le lien de l'élément précédent : */
                prev->next = p->next;
            }
            /* et on la détruit : */
            free(p->name);
            free(p->address);
            free(p);
        }
        return (p != NULL);
    }

    /* Simple fonction d'affichage. */
    void print_list(People const *lst)
    {
        Person const *p = *lst;
        int i = 1;
        while (p != NULL)
        {
            printf("Personne %d : %s (%s)\n", i, p->name, p->address);
            p = p->next;
            ++i;
        }
    }

    /* Fonction de destruction et de libération de la mémoire. */
    void destroy_list(People *lst)
    {
        while (*lst != NULL)
        {
            Person *p = *lst;
            *lst = p->next;
            free(p->name);
            free(p->address);
            free(p);
        }
        return ;
    }

    int main(void)
    {
        int op = 0;
        size_t s;
        char buffer[16];
        char name[256];
        char address[256];
        /* Crée une liste de personne : */
        People p;
        init_list(&p);
```

```
    /* Utilise la liste : */
    do
    {
        printf("Opération (0 = quitter, 1 = ajouter, 2 = supprimer) ?");
        fgets(buffer, 16, stdin);
        buffer[15] = 0;
        op = 3;
        sscanf(buffer, "%d", &op);
        switch (op)
        {
        case 0:
            break;
        case 1:
            printf("Nom : ");
            fgets(name, 256, stdin);   /* Lit le nom. */
            name[255] = 0;             /* Assure que le caractère nul
                                          terminal est écrit. */
            s = strlen(name);          /* Supprime l'éventuel saut de ligne. */
            if (name[s - 1] == '\n') name[s - 1] = 0;
            /* Même opération pour l'adresse : */
            printf("Adresse : ");
            fgets(address, 256, stdin);
            name[255] = 0;
            s = strlen(address);
            if (address[s - 1] == '\n') address[s - 1] = 0;
            add_person(&p, name, address);
            break;
        case 2:
            printf("Nom : ");
            fgets(name, 256, stdin);
            name[255] = 0;
            s = strlen(name);
            if (name[s - 1] == '\n') name[s - 1] = 0;
            if (remove_person(&p, name) == 0)
            {
                printf("Personne inconnue.\n");
            }
            break;
        default:
           printf("Opération invalide\n");
           break;
        }
        if (op != 0) print_list(&p);
    } while (op != 0);
    /* Détruit la liste : */
    destroy_list(&p);
    return EXIT_SUCCESS;
}
```

Note : Comme vous pouvez le constater, la lecture des chaînes de caractères saisies par l'utilisateur est réalisée au moyen de la fonction `fgets` de la bibliothèque C standard. Cette fonction permet de lire une ligne complète sur le flux spécifié en troisième paramètre, et de stocker le résultat dans la chaîne de caractères fournie en premier paramètre. Elle ne lira pas plus de caractères que le nombre indiqué en deuxième paramètre, ce qui permet de contrôler la taille des lignes saisies par l'utilisateur. La fonction `fgets` nécessite malheureusement quelques traitements supplémentaires avant de pouvoir utiliser la chaîne de caractères lue, car elle n'écrit pas le caractère nul terminal de la chaîne C si le nombre maximal de caractères à lire est atteint,

et elle stocke le caractère de saut de ligne en fin de ligne si ce nombre n'est pas atteint. Il est donc nécessaire de s'assurer que la ligne se termine bien par un caractère nul terminal d'une part, et de supprimer le caractère de saut de ligne s'il n'est pas essentiel d'autre part. Ces traitements constituent également un bon exemple de manipulation des pointeurs et des chaînes de caractères.

Ce programme n'interdit pas les définitions multiples de personnes ayant le même nom. Il n'interdit pas non plus la définition de personnes anonymes. Le lecteur pourra essayer de corriger ces petits défauts à titre d'exercice, afin de s'assurer que les notions de pointeur sont bien assimilées. Rappelons que les pointeurs sont une notion essentielle en C et qu'il faut être donc parfaitement familiarisé avec eux.

4.11.2. Allocation dynamique en C++

En plus des fonctions `malloc` et `free` du C, le C++ fournit d'autres moyens pour allouer et restituer la mémoire. Pour cela, il dispose d'opérateurs spécifiques : `new`, `delete`, `new[]` et `delete[]`. La syntaxe de ces opérateurs est respectivement la suivante :

```
new type
delete pointeur
new type[taille]
delete[] pointeur
```

Les deux opérateurs `new` et `new[]` permettent d'allouer de la mémoire, et les deux opérateurs `delete` et `delete[]` de la restituer.

La syntaxe de `new` est très simple, il suffit de faire suivre le mot clé `new` du type de la variable à allouer, et l'opérateur renvoie directement un pointeur sur cette variable avec le bon type. Il n'est donc plus nécessaire d'effectuer un transtypage après l'allocation, comme c'était le cas pour la fonction `malloc`. Par exemple, l'allocation d'un entier se fait comme suit :

```
int *pi = new int;  // Équivalent à (int *) malloc(sizeof(int)).
```

La syntaxe de `delete` est encore plus simple, puisqu'il suffit de faire suivre le mot clé `delete` du pointeur sur la zone mémoire à libérer :

```
delete pi;          // Équivalent à free(pi);
```

Les opérateurs `new[]` et `delete[]` sont utilisés pour allouer et restituer la mémoire pour les types tableaux. Ce ne sont pas les mêmes opérateurs que `new` et `delete`, et la mémoire allouée par les uns ne peut pas être libérée par les autres. Si la syntaxe de `delete[]` est la même que celle de delete, l'emploi de l'opérateur `new[]` nécessite de donner la taille du tableau à allouer. Ainsi, on pourra créer un tableau de 10000 entiers de la manière suivante :

```
int *Tableau=new int[10000];
```

et détruire ce tableau de la manière suivante :

```
delete[] Tableau;
```

L'opérateur `new[]` permet également d'allouer des tableaux à plusieurs dimensions. Pour cela, il suffit de spécifier les tailles des différentes dimensions à la suite du type de donnée des élements du tableau, exactement comme lorsque l'on crée un tableau statiquement. Toutefois, seule la première dimension du tableau peut être variable, et les dimensions deux et suivantes doivent avoir une taille entière positive et constante. Par exemple, seule la deuxième ligne de l'exemple qui suit est une allocation dynamique de tableau valide :

```
int i=5, j=3;
int (*pi1)[3] = new int[i][3]; // Alloue un tableau de i lignes de trois entiers.
int (*pi2)[3] = new int[i][j]; // Illégal, j n'est pas constant.
```

Si l'on désire réellement avoir des tableaux dont plusieurs dimensions sont de taille variable, on devra allouer un tableau de pointeurs et, pour chaque ligne de ce tableau, allouer un autre tableau à la main.

> **Note :** Il est important d'utiliser l'opérateur `delete[]` avec les pointeurs renvoyés par l'opérateur `new[]` et l'opérateur `delete` avec les pointeurs renvoyés par `new`. De plus, on ne devra pas non plus mélanger les mécanismes d'allocation mémoire du C et du C++ (utiliser `delete` sur un pointeur renvoyé par `malloc` par exemple). En effet, le compilateur peut allouer une quantité de mémoire supérieure à celle demandée par le programme afin de stocker des données qui lui permettent de gérer la mémoire. Ces données peuvent être interprétées différemment pour chacune des méthodes d'allocation, si bien qu'une utilisation erronée peut entraîner soit la perte des blocs de mémoire, soit une erreur, soit un plantage.

L'opérateur `new[]` alloue la mémoire et crée les objets dans l'ordre croissant des adresses. Inversement, l'opérateur `delete[]` détruit les objets du tableau dans l'ordre décroissant des adresses avant de libérer la mémoire.

La manière dont les objets sont construits et détruits par les opérateurs `new` et `new[]` dépend de leur nature. S'il s'agit de types de base du langage ou de structures simples, aucune initialisation particulière n'est faite. La valeur des objets ainsi créés est donc indéfinie, et il faudra réaliser l'initialisation soi-même. Si, en revanche, les objets créés sont des instances de classes C++, le constructeur de ces classes sera automatiquement appelé lors de leur initialisation. C'est pour cette raison que l'on devra, de manière générale, préférer les opérateurs C++ d'allocation et de désallocation de la mémoire aux fonctions `malloc` et `free` du C. Ces opérateurs ont de plus l'avantage de permettre un meilleur contrôle des types de données et d'éviter un transtypage. Les notions de classe et de constructeur seront présentées en détail dans le chapitre traitant de la couche objet du C++.

Lorsqu'il n'y a pas assez de mémoire disponible, les opérateurs `new` et `new[]` peuvent se comporter de deux manières selon l'implémentation. Le comportement le plus répandu est de renvoyer un pointeur nul. Cependant, la norme C++ indique un comportement différent : si l'opérateur `new` manque de mémoire, il doit appeler un gestionnaire d'erreur. Ce gestionnaire ne prend aucun paramètre et ne renvoie rien. Selon le comportement de ce gestionnaire d'erreur, plusieurs actions peuvent être faites :

- soit ce gestionnaire peut corriger l'erreur d'allocation et rendre la main à l'opérateur `new` (le programme n'est donc pas terminé), qui effectue une nouvelle tentative pour allouer la mémoire demandée ;
- soit il ne peut rien faire. Dans ce cas, il peut mettre fin à l'exécution du programme ou lancer une exception `std::bad_alloc`, qui remonte alors jusqu'à la fonction appelant l'opérateur `new`.

C'est le comportement du gestionnaire installé par défaut dans les implémentations conformes à la norme.

L'opérateur `new` est donc susceptible de lancer une exception `std::bad_alloc`. Voir le Chapitre 8 pour plus de détails à ce sujet.

Il est possible de remplacer le gestionnaire d'erreur appelé par l'opérateur `new` à l'aide de la fonction `std::set_new_handler`, déclarée dans le fichier d'en-tête `new`. Cette fonction attend en paramètre un pointeur sur une fonction qui ne prend aucun paramètre et ne renvoie rien. Elle renvoie l'adresse du gestionnaire d'erreur précédent.

> **Note :** La fonction `std::set_new_handler` et la classe std::bad_alloc font partie de la bibliothèque standard C++. Comme leurs noms l'indiquent, ils sont déclarés dans l'espace de nommage `std::`, qui est réservé pour les fonctions et les classes de la bibliothèque standard. Voyez aussi le Chapitre 10 pour plus de détails sur les espaces de nommages. Si vous ne désirez pas utiliser les mécanismes des espaces de nommage, vous devrez inclure le fichier d'en-tête `new.h` au lieu de `new`.
>
> Attendez vous à ce qu'un jour, tous les compilateurs C++ lancent une exception en cas de manque de mémoire lors de l'appel à l'opérateur `new`, car c'est ce qu'impose la norme. Si vous ne désirez pas avoir à gérer les exceptions dans votre programme et continuer à recevoir un pointeur nul en cas de manque de mémoire, vous pouvez fournir un deuxième paramètre de type std::nothrow_t à l'opérateur `new`. La bibliothèque standard définit l'objet constant `std::nothrow` à cet usage.

Les opérateurs `delete` et `delete[]` peuvent parfaitement être appelés avec un pointeur nul en paramètre. Dans ce cas, ils ne font rien et redonnent la main immédiatement à l'appelant. Il n'est donc pas nécessaire de tester la non nullité des pointeurs sur les objets que l'on désire détruire avant d'appeler les opérateurs `delete` et `delete[]`.

4.12. Pointeurs et références de fonctions

4.12.1. Pointeurs de fonctions

Il est possible de faire des pointeurs de fonctions. Un pointeur de fonction contient l'adresse du début du code binaire constituant la fonction. Il est possible d'appeler une fonction dont l'adresse est contenue dans un pointeur de fonction avec l'opérateur d'indirection `*`.

Pour déclarer un pointeur de fonction, il suffit de considérer les fonctions comme des variables. Leur déclaration est identique à celle des tableaux, en remplaçant les crochets par des parenthèses :

```
type (*identificateur)(paramètres);
```

où `type` est le type de la valeur renvoyée par la fonction, `identificateur` est le nom du pointeur de la fonction et `paramètres` est la liste des types des variables que la fonction attend comme paramètres, séparés par des virgules.

Exemple 4-13. Déclaration de pointeur de fonction

```
int (*pf)(int, int);   /* Déclare un pointeur de fonction. */
```

`pf` est un pointeur de fonction attendant comme paramètres deux entiers et renvoyant un entier.

Il est possible d'utiliser `typedef` pour créer un alias du type pointeur de fonction :

```
typedef int (*PtrFonct)(int, int);
PtrFonct pf;
```

`PtrFonct` est le type des pointeurs de fonctions.

Si `f` est une fonction répondant à ces critères, on peut alors initialiser `pf` avec l'adresse de `f`. De même, on peut appeler la fonction pointée par `pf` avec l'opérateur d'indirection.

Exemple 4-14. Déréférencement de pointeur de fonction

```
#include <stdlib.h>
#include <stdio.h>

int f(int i, int j)   /* Définit une fonction. */
{
    return i+j;
}

int (*pf)(int, int);  /* Déclare un pointeur de fonction. */

int main(void)
{
    int l, m;         /* Déclare deux entiers. */
    pf = &f;          /* Initialise pf avec l'adresse de la fonction f. */
    printf("Entrez le premier entier : ");
    scanf("%u",&l);   /* Initialise les deux entiers. */
    printf("\nEntrez le deuxième entier : ");
    scanf("%u",&m);

/* Utilise le pointeur pf pour appeler la fonction f
   et affiche le résultat : */

    printf("\nLeur somme est de : %u\n", (*pf)(l,m));
    return EXIT_SUCCESS;
}
```

L'intérêt des pointeurs de fonction est de permettre l'appel d'une fonction parmi un éventail de fonctions au choix.

Par exemple, il est possible de faire un tableau de pointeurs de fonctions et d'appeler la fonction dont on connaît l'indice de son pointeur dans le tableau.

Exemple 4-15. Application des pointeurs de fonctions

```
#include <stdlib.h>
#include <stdio.h>

/* Définit plusieurs fonctions travaillant sur des entiers : */

int somme(int i, int j)
{
    return i+j;
```

```
}

int multiplication(int i, int j)
{
    return i*j;
}

int quotient(int i, int j)
{
    return i/j;
}

int modulo(int i, int j)
{
    return i%j;
}

typedef int (*fptr)(int, int);
fptr ftab[4];

int main(void)
{
    int i,j,n;
    ftab[0]=&somme;          /* Initialise le tableau de pointeur */
    ftab[1]=&multiplication; /* de fonctions. */
    ftab[2]=&quotient;
    ftab[3]=&modulo;
    printf("Entrez le premier entier : ");
    scanf("%u",&i);          /* Demande les deux entiers i et j. */
    printf("\nEntrez le deuxième entier : ");
    scanf("%u",&j);
    printf("\nEntrez la fonction : ");
    scanf("%u",&n);          /* Demande la fonction à appeler. */
    if (n < 4)
        printf("\nRésultat : %u.\n", (*(ftab[n]))(i,j) );
    else
        printf("\nMauvais numéro de fonction.\n");
    return EXIT_SUCCESS;
}
```

4.12.2. Références de fonctions

Les références de fonctions sont acceptées en C++. Cependant, leur usage est assez limité. Elles permettent parfois de simplifier les écritures dans les manipulations de pointeurs de fonctions. Mais comme il n'est pas possible de définir des tableaux de références, le programme d'exemple donné ci-dessus ne peut pas être récrit avec des références.

Les références de fonctions peuvent malgré tout être utilisées à profit dans le passage des fonctions en paramètre dans une autre fonction. Par exemple :

```
#include <stdio.h>

// Fonction de comparaison de deux entiers :

int compare(int i, int j)
```

```
{
    if (i<j) return -1;
    else if (i>j) return 1;
    else return 0;
}

// Fonction utilisant une fonction en tant que paramètre :

void trie(int tableau[], int taille, int (&fcomp)(int, int))
{
    // Effectue le tri de tableau avec la fonction fcomp.
    // Cette fonction peut être appelée comme toute les autres
    // fonctions :
    printf("%d", fcomp(2,3));
      ⋮
    return ;
}

int main(void)
{
    int t[3]={1,5,2};
    trie(t, 3, compare);   // Passage de compare() en paramètre.
    return 0;
}
```

4.13. Paramètres de la fonction main - ligne de commande

L'appel d'un programme se fait normalement avec la syntaxe suivante :

```
nom param1 param2 [...]
```

où `nom` est le nom du programme à appeler et `param1`, etc. sont les paramètres de la ligne de commande. De plus, le programme appelé peut renvoyer un code d'erreur au programme appelant (soit le système d'exploitation, soit un autre programme). Ce code d'erreur est en général `0` quand le programme s'est déroulé correctement. Toute autre valeur indique qu'une erreur s'est produite en cours d'exécution.

La valeur du code d'erreur est renvoyée par la fonction `main`. Le code d'erreur doit toujours être un entier. La fonction `main` peut donc (et même normalement doit) être de type entier :

```
int main(void) ...
```

Les paramètres de la ligne de commandes peuvent être récupérés par la fonction `main`. Si vous désirez les récupérer, la fonction `main` doit attendre deux paramètres :

- le premier est un entier, qui représente le nombre de paramètres ;

- le deuxième est un tableau de chaînes de caractères (donc en fait un tableau de pointeurs, ou encore un pointeur de pointeurs de caractères).

Les paramètres se récupèrent avec ce tableau. Le premier élément pointe toujours sur la chaîne donnant le nom du programme. Les autres éléments pointent sur les paramètres de la ligne de commande.

Exemple 4-16. Récupération de la ligne de commande

```
#include <stdlib.h>
#include <stdio.h>

int main(int n, char *params[])  /* Fonction principale. */
{
    int i;

    /* Affiche le nom du programme : */
    printf("Nom du programme : %s.\n",params[0]);

    /* Affiche la ligne de commande : */
    for (i=1; i<n; ++i)
        printf("Argument %d : %s.\n",i, params[i]);
    return EXIT_SUCCESS;   /* Tout s'est bien passé : on renvoie 0 ! */
}
```

4.14. DANGER

Les pointeurs sont, comme on l'a vu, très utilisés en C/C++. Il faut donc bien savoir les manipuler.

Mais ils sont très dangereux, car ils permettent d'accéder à n'importe quelle zone mémoire, s'ils ne sont pas correctement initialisés. Dans ce cas, ils pointent n'importe où. Accéder à la mémoire avec un pointeur non initialisé peut altérer soit les données du programme, soit le code du programme lui-même, soit le code d'un autre programme ou celui du système d'exploitation. Cela conduit dans la majorité des cas au plantage du programme, et parfois au plantage de l'ordinateur si le système ne dispose pas de mécanismes de protection efficaces.

VEILLEZ À TOUJOURS INITIALISER LES POINTEURS QUE VOUS UTILISEZ.

Pour initialiser un pointeur qui ne pointe sur rien (c'est le cas lorsque la variable pointée n'est pas encore créée ou lorsqu'elle est inconnue lors de la déclaration du pointeur), on utilisera le pointeur prédéfini `NULL`.

VÉRIFIEZ QUE TOUTE DEMANDE D'ALLOCATION MÉMOIRE A ÉTÉ SATISFAITE.

La fonction `malloc` renvoie le pointeur `NULL` lorsqu'il n'y a plus ou pas assez de mémoire. Le comportement des opérateurs `new` et `new[]` est différent. Théoriquement, ils doivent lancer une exception

si la demande d'allocation mémoire n'a pas pu être satisfaite. Cependant, certains compilateurs font en sorte qu'ils renvoient le pointeur nul du type de l'objet à créer.

S'ils renvoient une exception, le programme sera arrêté si aucun traitement particulier n'est fait. Bien entendu, le programme peut traiter cette exception s'il le désire, mais en général, il n'y a pas grand chose à faire en cas de manque de mémoire. Vous pouvez consulter le chapitre traitant des exceptions pour plus de détails à ce sujet.

Dans tous les cas,

```
LORSQU'ON UTILISE UN POINTEUR, IL FAUT VÉRIFIER S'IL EST VALIDE
```

(par un test avec `NULL` ou le pointeur nul, ou en analysant l'algorithme). Cette vérification inclut le test de débordement lors des accès aux chaînes de caractères et aux tableaux. Cela est extrêmement important lorsque l'on manipule des données provenant de l'extérieur du programme, car on ne peut dans ce cas pas supposer que ces données sont valides.

Chapitre 5. Le préprocesseur C

5.1. Définition

Le *préprocesseur* est un programme qui analyse un fichier texte et qui lui fait subir certaines transformations. Ces transformations peuvent être l'*inclusion d'un fichier*, la *suppression* d'une zone de texte ou le *remplacement* d'une zone de texte.

Le préprocesseur effectue ces opérations en suivant des ordres qu'il lit dans le fichier en cours d'analyse.

Il est appelé automatiquement par le compilateur, avant la compilation, pour traiter les fichiers à compiler.

5.2. Les directives du préprocesseur

Une *directive* est une commande pour le préprocesseur. Toutes les directives du préprocesseur commencent :

- en début de ligne ;
- par un signe dièse (`#`).

Le préprocesseur dispose de directives permettant d'inclure des fichiers, de définir des constantes de compilation, de supprimer conditionnellement des blocs de texte, et de générer des erreurs ou de modifier l'environnement de compilation.

5.2.1. Inclusion de fichier

L'inclusion de fichier permet de factoriser du texte commun à plusieurs autres fichiers (par exemple des déclarations de type, de constante, de fonction, etc.). Les déclarations des fonctions et des constantes sont ainsi généralement factorisées dans un fichier d'en-tête portant l'extension `.h` pour « header », fichier d'en-tête de programme).

Par exemple, nous avons déjà vu les fichiers d'en-tête `stdlib.h`, `stdio.h` et `string.h` dans les chapitres précédents. Ce sont vraissemblablement les fichiers d'en-tête de la bibliothèque C les plus couramment utilisés. Si vous ouvrez le fichier `stdio.h`, vous y verrez la déclaration de toutes les fonctions et de tous les types de la bibliothèque d'entrée - sortie standard (notez qu'elles sont peut-être définies dans d'autres fichiers d'en-tête, eux-mêmes inclus par ce fichier). De même, vous trouverez sans doute les déclarations des fonctions `malloc` et `free` dans le fichier d'en-tête `stdlib.h`.

La syntaxe générale pour la directive d'inclusion de fichier est la suivante :

```
#include "fichier"
```

ou :

```
#include <fichier>
```

`fichier` est le nom du fichier à inclure. Lorsque son nom est entre guillemets, le fichier spécifié est recherché dans le répertoire courant (normalement le répertoire du programme). S'il est encadré de crochets, il est recherché d'abord dans les répertoires spécifiés en ligne de commande avec l'option `-I`, puis dans les répertoires du chemin de recherche des en-têtes du système (ces règles ne sont pas fixes, elles ne sont pas normalisées).

Le fichier inclus est traité lui aussi par le préprocesseur.

5.2.2. Constantes de compilation et remplacement de texte

Le préprocesseur permet de définir des identificateurs qui, utilisés dans le programme, seront remplacés textuellement par leur valeur. La définition de ces identificateurs suit la syntaxe suivante :

```
#define identificateur texte
```

où `identificateur` est l'identificateur qui sera utilisé dans la suite du programme, et `texte` sera le texte de remplacement que le préprocesseur utilisera. Le texte de remplacement est facultatif (dans ce cas, c'est le texte vide). À chaque fois que l'identificateur `identificateur` sera rencontré par le préprocesseur, il sera remplacé par le texte `texte` dans toute la suite du programme.

Cette commande est couramment utilisée pour définir des *constantes de compilation*, c'est-à-dire des constantes qui décrivent les paramètres de la plateforme pour laquelle le programme est compilé. Ces constantes permettent de réaliser des *compilations conditionnelles*, c'est-à-dire de modifier le comportement du programme en fonction de paramètres définis lors de sa compilation. Elle est également utilisée pour remplacer des identificateurs du programme par d'autres identificateurs, par exemple afin de tester plusieurs versions d'une même fonction sans modifier tout le programme.

Exemple 5-1. Définition de constantes de compilation

```
#define UNIX_SOURCE
#define POSIX_VERSION 1001
```

Dans cet exemple, l'identificateur `UNIX_SOURCE` sera défini dans toute la suite du programme, et la constante de compilation `POSIX_VERSION` sera remplacée par `1001` partout où elle apparaîtra.

> **Note :** On fera une distinction bien nette entre les constantes de compilation définies avec la directive `#define` du préprocesseur et les constantes définies avec le mot clé `const`. En effet, les constantes littérales ne réservent pas de mémoire. Ce sont des valeurs immédiates, définies par le compilateur. En revanche, les variables de classe de stockage `const` peuvent malgré tout avoir une place mémoire réservée. Ce peut par exemple être le cas si l'on manipule leur adresse ou s'il ne s'agit pas de vraies constantes, par exemple si elles peuvent être modifiées par l'environnement (dans ce cas, elles doivent être déclarées avec la classe de stockage `volatile`). Ce sont donc plus des variables accessibles en lecture seule que des constantes. On ne pourra jamais supposer qu'une variable ne change pas de valeur sous prétexte qu'elle a la classe de stockage `const`, alors qu'évidemment, une constante littérale déclarée avec la directive `#define` du préprocesseur conservera toujours sa valeur (pourvu qu'on ne la redéfinisse pas). Par ailleurs, les constantes littérales n'ont pas de type, ce qui peut être très gênant et source d'erreur. On réservera donc leur emploi uniquement pour les constantes de compilation, et on préférera le mot clé `const` pour toutes les autres constantes du programme.

Le préprocesseur définit un certain nombre de constantes de compilation automatiquement. Ce sont les suivantes :

- `__LINE__` : donne le numéro de la ligne courante ;
- `__FILE__` : donne le nom du fichier courant ;
- `__DATE__` : renvoie la date du traitement du fichier par le préprocesseur ;
- `__TIME__` : renvoie l'heure du traitement du fichier par le préprocesseur ;
- `__cplusplus` : définie uniquement dans le cas d'une compilation C++. Sa valeur doit être `199711L` pour les compilateurs compatibles avec le projet de norme du 2 décembre 1996. En pratique, sa valeur est dépendante de l'implémentation utilisée, mais on pourra utiliser cette chaîne de remplacement pour distinguer les parties de code écrites en C++ de celles écrites en C.

Note : Si `__FILE__`, `__DATE__`, `__TIME__` et `__cplusplus` sont bien des constantes pour un fichier donné, ce n'est pas le cas de `__LINE__`. En effet, cette dernière « constante » change bien évidemment de valeur à chaque ligne. On peut considérer qu'elle est redéfinie automatiquement par le préprocesseur à chaque début de ligne.

5.2.3. Compilation conditionnelle

La définition des identificateurs et des constantes de compilation est très utilisée pour effectuer ce que l'on appelle la *compilation conditionnelle*. La compilation conditionnelle consiste à remplacer certaines portions de code source par d'autres, en fonction de la présence ou de la valeur de constantes de compilation. Cela est réalisable à l'aide des directives de compilation conditionnelle, dont la plus courante est sans doute `#ifdef` :

```
#ifdef identificateur
   ⋮
#endif
```

Dans l'exemple précédent, le texte compris entre le `#ifdef` (c'est-à-dire « if defined ») et le `#endif` est laissé tel quel si l'identificateur `identificateur` est connu du préprocesseur. Sinon, il est supprimé. L'identificateur peut être déclaré en utilisant simplement la commande `#define` vue précédemment.

Il existe d'autres directives de compilation conditionnelle :

```
#ifndef      (if not defined ...)
#elif        (sinon, si ... )
#if          (si ... )
```

La directive `#if` attend en paramètre une expression constante. Le texte qui la suit est inclus dans le fichier si et seulement si cette expression est non nulle. Par exemple :

```
#if (__cplusplus==199711L)
   ⋮
#endif
```

permet d'inclure un morceau de code C++ strictement conforme à la norme décrite dans le projet de norme du 2 décembre 1996.

Une autre application courante des directives de compilation est la protection des fichiers d'en-tête contre les inclusions multiples :

```
#ifndef DejaLa
#define DejaLa

Texte à n'inclure qu'une seule fois au plus.

#endif
```

Cela permet d'éviter que le texte soit inclus plusieurs fois, à la suite de plusieurs appels de `#include`. En effet, au premier appel, `DejaLa` n'est pas connu du préprocesseur. Il est donc déclaré et le texte est inclus. Lors de tout autre appel ultérieur, `DejaLa` existe, et le texte n'est pas inclus. Ce genre d'écriture se rencontre dans les fichiers d'en-tête, pour lesquels en général on ne veut pas qu'une inclusion multiple ait lieu.

5.2.4. Autres directives

Le préprocesseur est capable d'effectuer d'autres actions que l'inclusion et la suppression de texte. Les directives qui permettent d'effectuer ces actions sont indiquées ci-dessous :

- `#` : ne fait rien (directive nulle) ;
- `#error message` : permet de stopper la compilation en affichant le message d'erreur donné en paramètre ;
- `#line numéro [fichier]` : permet de changer le numéro de ligne courant et le nom du fichier courant lors de la compilation ;
- `#pragma texte` : permet de donner des ordres spécifiques à une l'implémentation du compilateur tout en conservant la portabilité du programme. Toute implémentation qui ne reconnaît pas un ordre donné dans une directive `#pragma` doit l'ignorer pour éviter des messages d'erreurs. Le format des ordres que l'on peut spécifier à l'aide de la directive `#pragma` n'est pas normalisé et dépend de chaque compilateur.

5.3. Les macros

Le préprocesseur peut, lors du mécanisme de remplacement de texte, utiliser des paramètres fournis à l'identificateur à remplacer. Ces paramètres sont alors replacés sans modification dans le texte de remplacement. Le texte de remplacement est alors appelé *macro*.

La syntaxe des macros est la suivante :

```
#define macro(paramètre[, paramètre [...]]) définition
```

Exemple 5-2. Macros MIN et MAX

```
#define MAX(x,y) ((x)>(y)?(x):(y))
#define MIN(x,y) ((x)<(y)?(x):(y))
```

Note : Pour poursuivre une définition sur la ligne suivante, terminez la ligne courante par le signe '\'.

Le mécanisme des macros permet de faire l'équivalent de fonctions générales, qui fonctionnent pour tous les types. Ainsi, la macro `MAX` renvoie le maximum de ses deux paramètres, qu'ils soient entiers, longs ou réels. Cependant, on prendra garde au fait que les paramètres passés à une macro sont évalués par celle-ci à chaque fois qu'ils sont utilisés dans la définition de la macro. Cela peut poser des problèmes de performances ou, pire, provoquer des effets de bords indésirables. Par exemple, l'utilisation suivante de la macro `MIN` :

```
MIN(f(3), 5)
```

provoque le remplacement suivant :

```
((f(3))<(5))?(f(3)):(5))
```

soit deux appels de la fonction `f` si `f(3)` est inférieur à `5`, et un seul appel sinon. Si la fonction `f` ainsi appelée modifie des variables globales, le résultat de la macro ne sera certainement pas celui attendu, puisque le nombre d'appels est variable pour une même expression. On évitera donc, autant que faire se peut, d'utiliser des expressions ayant des effets de bords en paramètres d'une macro. Les écritures du type :

```
MIN(++i, j)
```

sont donc à prohiber.

Il est possible de faire des macros à nombre d'arguments variables depuis la norme C99. Ces macros se déclarent comme des fonctions à nombre d'arguments variables, ces arguments pouvant être récupérés en blocs grâce à la macro préfédinie `__VA_ARGS__` et utilisés en lieu et place de ces arguments dans une fonction à nombre d'arguments variable classique.

Exemple 5-3. Macro à nombre d'arguments variable

```
#include <stdlib.h>
#include <stdio.h>

/* Définition d'une macro d'écriture formatée sur le flux
   d'erreur standard : */
#define printerrf(...) \
    { \
        fprintf(stderr, __VA_ARGS__); \
    }

int main(void)
{
    printerrf("Erreur %d (%s)\n", 5, "erreur d'entrée / sortie");
    return EXIT_SUCCESS;
}
```

On mettra toujours des parenthèses autour des paramètres de la macro. En effet, ces paramètres peuvent être des expressions composées, qui doivent être calculées complètement avant d'être utilisées dans la macro. Les parenthèses forcent ce calcul. Si on ne les met pas, les règles de priorités

peuvent générer une erreur de logique dans la macro elle-même. De même, on entourera de parenthèses les macros renvoyant une valeur, afin de forcer leur évaluation complète avant toute utilisation dans une autre expression. Par exemple :

```
#define mul(x,y) x*y
```

est une macro fausse. La ligne :

```
mul(2+3,5+9)
```

sera remplacée par :

```
2+3*5+9
```

ce qui vaut `26`, et non pas `70` comme on l'aurait attendu. La bonne macro est :

```
#define mul(x,y) ((x)*(y))
```

car elle donne le texte suivant :

```
((2+3)*(5+9))
```

et le résultat est correct. De même, la macro :

```
#define add(x,y) (x)+(y)
```

est fausse, car l'expression suivante :

```
add(2,3)*5
```

est remplacée textuellement par :

```
(2)+(3)*5
```

dont le résultat est `17` et non `25` comme on l'aurait espéré. Cette macro doit donc se déclarer comme suit :

```
#define add(x,y) ((x)+(y))
```

Ainsi, les parenthèses assurent un comportement cohérent de la macro. Comme on le voit, les parenthèses peuvent alourdir les définitions des macros, mais elles sont absolument nécessaires.

Le résultat du remplacement d'une macro par sa définition est, lui aussi, soumis au préprocesseur. Par conséquent, une macro peut utiliser une autre macro ou une constante définie avec `#define`. Cependant, ce mécanisme est limité aux macros qui n'ont pas encore été remplacées afin d'éviter une récursion infinie du préprocesseur. Par exemple :

```
#define toto(x) toto((x)+1)
```

définit la macro `toto`. Si plus loin on utilise « `toto(3)` », le texte de remplacement final sera « `toto((3)+1)` » et non pas l'expression infinie « `(...(((3)+1)+1...)+1` ».

Le préprocesseur définit automatiquement la macro `defined`, qui permet de tester si un identificateur est connu du préprocesseur. Sa syntaxe est la suivante :

```
defined(identificateur)
```

La valeur de cette macro est `1` si l'identificateur existe, `0` sinon. Elle est utilisée principalement avec la directive `#if`. Il est donc équivalent d'écrire :

```
#if defined(identificateur)
   ⋮
#endif
```

et :

```
#ifdef identificateur
   ⋮
#endif
```

Cependant, `defined` permet l'écriture d'expressions plus complexes que la directive `#if`.

5.4. Manipulation de chaînes de caractères dans les macros

Le préprocesseur permet d'effectuer des opérations sur les chaînes de caractères. Tout argument de macro peut être transformé en chaîne de caractères dans la définition de la macro s'il est précédé du signe `#`. Par exemple, la macro suivante :

```
#define CHAINE(s) #s
```

transforme son argument en chaîne de caractères. Par exemple :

```
CHAINE(2+3)
```

devient :

```
"2+3"
```

Lors de la transformation de l'argument, toute occurrence des caractères `"` et `\` est transformée respectivement en `\"` et `\\` pour conserver ces caractères dans la chaîne de caractères de remplacement.

Le préprocesseur permet également la concaténation de texte grâce à l'opérateur `##`. Les arguments de la macro qui sont séparés par cet opérateur sont concaténés (sans être transformés en chaînes de caractères cependant). Par exemple, la macro suivante :

```
#define NOMBRE(chiffre1,chiffre2) chiffre1##chiffre2
```

permet de construire un nombre à deux chiffres :

```
NOMBRE(2,3)
```

est remplacé par le nombre décimal `23`. Le résultat de la concaténation est ensuite analysé pour d'éventuels remplacements additionnels par le préprocesseur.

5.5. Les trigraphes

Le jeu de caractères utilisé par le langage C++ comprend toutes les lettres en majuscules et en minuscules, tous les chiffres et les caractères suivants :

```
. , ; : ! ? " ' + - ^ * % = &  | ~ _ # / \ { } [ ] () < >
```

Malheureusement, certains environnements sont incapables de gérer quelques-uns de ces caractères. C'est pour résoudre ce problème que les *trigraphes* ont été créés.

Les trigraphes sont des séquences de trois caractères commençant par deux points d'interrogations. Ils permettent de remplacer les caractères qui ne sont pas accessibles sur tous les environnements. Vous n'utiliserez donc sans doute jamais les trigraphes, à moins d'y être forcé. Les trigraphes disponibles sont définis ci-dessous :

Tableau 5-1. Trigraphes

Trigraphe	Caractère de remplacement
??=	#
??/	\
??'	^
??(	[
??)	]
??!	|
??<	{
??>	}
??-	~

Chapitre 6. Modularité des programmes et génération des binaires

La *modularité* est le fait, pour un programme, d'être écrit en plusieurs morceaux relativement indépendants les uns des autres. La modularité a d'énormes avantages lors du développement d'un programme. Cependant, elle implique un processus de génération de l'exécutable assez complexe. Dans ce chapitre, nous allons voir l'intérêt de la modularité, les différentes étapes qui permettent la génération de l'exécutable et l'influence de ces étapes sur la syntaxe du langage.

6.1. Pourquoi faire une programmation modulaire ?

Ce qui coûte le plus cher en informatique, c'est le développement de logiciel, pas le matériel. En effet, développer un logiciel demande du temps, de la main d'œuvre qualifiée, et n'est pas facile (il y a toujours des erreurs). Les logiciels développés sont souvent spécifiques à un type de problème donné, alors que le matériel, conçu pour être générique, est utilisable dans de nombreuses situations diverses et bénéficie d'économies d'échelle qui amortissent les frais de recherche et de production. Autrement dit, les coûts spécifiques ont été déplacés, à tord ou à raison, du matériel vers le logiciel. Donc pour chaque problème, il faut faire un logiciel qui le résoud.

Les coûts de réalisation d'un logiciel se situent certainement majoritairement dans les tâches de spécifications et de conception. Les tâches de plus bas niveau, tels que le codage et la programmation, sont généralement plus techniques et sont plus faciles à réaliser.

Quasiment tout le monde cherche donc à optimiser chacune de ces tâches et à en diminuer les coûts. Pour ce faire, une branche de l'informatique a été développée : le *génie logiciel*. Le génie logiciel donne les grands principes à appliquer lors de la réalisation d'un programme, de la conception à la distribution, et sur toute la durée de vie du projet. Ce sujet dépasse largement le cadre de ce cours, aussi je ne parlerais que de l'aspect codage seul, qui est bien entendu plus technique, et qui concerne le programmeur C/C++.

Au niveau du codage, le plus important est la programmation modulaire. Les idées qui en sont à la base sont les suivantes :

- diviser le travail en plusieurs équipes ;
- créer des morceaux de programme indépendants de la problématique globale, donc réutilisables pour d'autres logiciels ;
- supprimer les risques d'erreurs qu'on avait en reprogrammant ces morceaux à chaque fois.

Je tiens à préciser que les principes de la programmation modulaire ne s'appliquent pas qu'aux programmes développés par des équipes de programmeurs, bien au contraire ! Ils s'appliquent aussi aux programmeurs individuels. En effet, comme la plupart des programmeurs individuels attaquent généralement les problèmes directement au niveau codage (alors qu'ils devraient le faire au niveau conception, voire de spécification des besoins et de délimitation du périmètre fonctionnel), les seules techniques de génie logiciel qu'ils peuvent appliquer sont les techniques de codage. De plus, il est plus facile de décomposer un problème en ses éléments, forcément plus simples, que de le traiter dans sa totalité (dixit Descartes), et une conception modulaire devient ainsi une aide.

Pour parvenir à ce but, il est indispensable de pouvoir découper un programme en sous-programmes indépendants, ou presque indépendants. Pour que chacun puisse travailler sur sa partie de programme

et que les problèmes résolus soient indépendants et réutilisables, il faut que ces morceaux de programme soient dans des fichiers séparés.

Pour pouvoir vérifier ces morceaux de programme, il faut que les compilateurs puissent les compiler indépendamment, sans avoir les autres fichiers du programme. Ainsi, le développement de chaque fichier peut se faire relativement indépendamment de celui des autres. Cependant, cette division du travail implique des opérations assez complexes pour générer l'exécutable.

6.2. Les différentes phases du processus de génération des exécutables

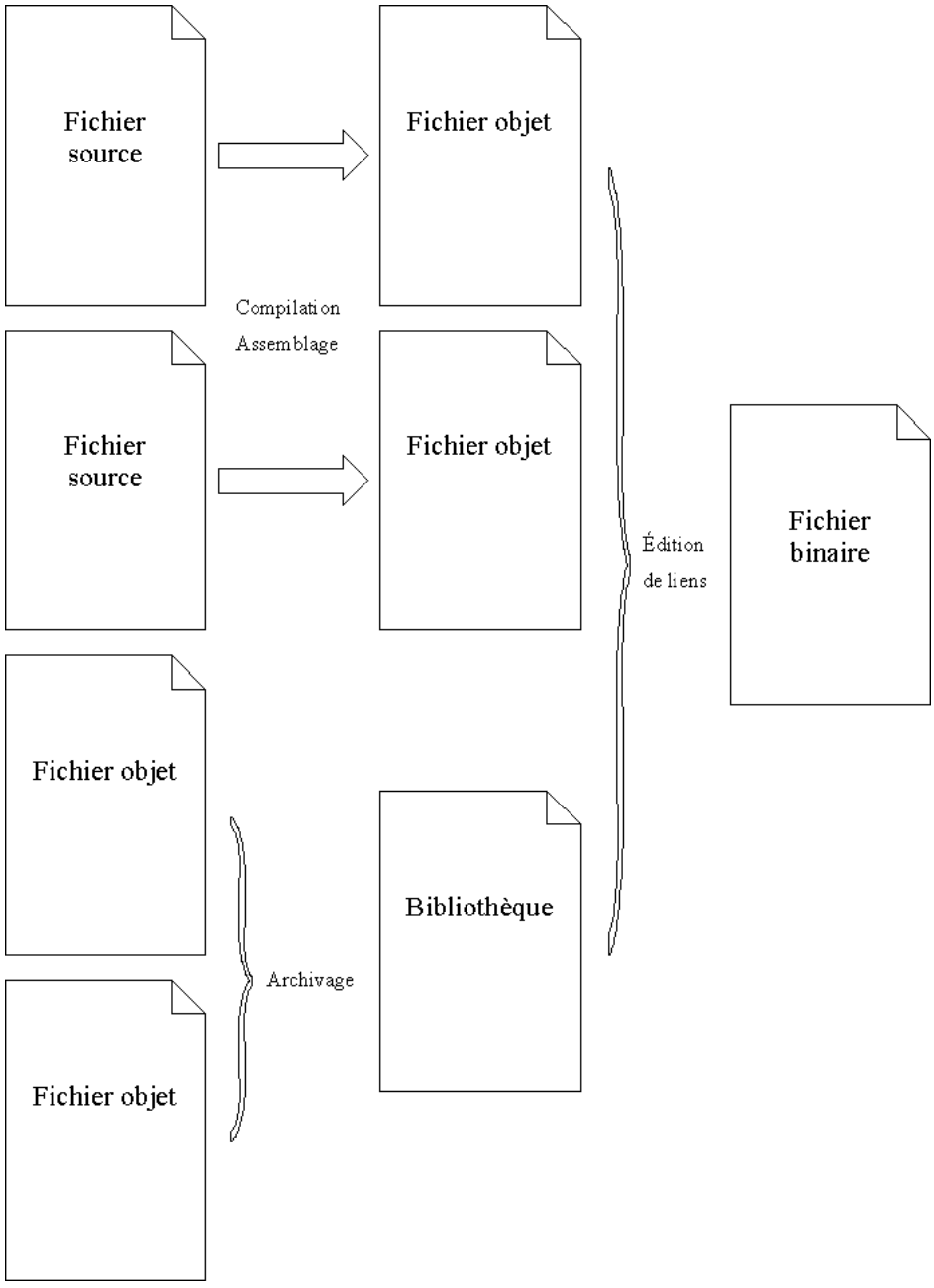
Les phases du processus qui conduisent à l'exécutable à partir des fichiers sources d'un programme sont décrites ci-dessous. Ces phases ne sont en général pas spécifiques au C++ et peuvent souvent être généralisées à tout langage compilé. Même si les différents outils de programmation peuvent les cacher, le processus de génération des exécutables se déroule toujours selon les principes qui suivent.

Au début de la génération de l'exécutable, on ne dispose que des fichiers sources du programme, écrit en C, C++ ou tout autre langage (ce qui suit n'est pas spécifique au C/C++). En général, la première étape est le traitement des fichiers sources avant compilation. Dans le cas du C et du C++, il s'agit des opérations effectuées par le *préprocesseur* (remplacement de macros, suppression de texte, inclusion de fichiers...).

Vient ensuite la *compilation séparée*, qui est le fait de compiler séparément les fichiers sources. Le résultat de la compilation d'un *fichier source* est généralement un fichier en *assembleur*, c'est-à-dire le langage décrivant les instructions du microprocesseur de la machine cible pour laquelle le programme est destiné. Les fichiers en assembleur peuvent être traduits directement en ce que l'on appelle des *fichiers objets*. Les fichiers objets contiennent la traduction du code assembleur en *langage machine*. Ils contiennent aussi d'autres informations, par exemple les données initialisées et les informations qui seront utilisées lors de la création du fichier exécutable à partir de tous les fichiers objets générés. Les fichiers objets peuvent être regroupés en bibliothèques statiques, afin de rassembler un certain nombre de fonctionnalités qui seront utilisées ultérieurement.

Enfin, l'étape finale du processus de compilation est le regroupement de toutes les données et de tout le code des fichiers objets du programme et des bibliothèques (fonctions de la bibliothèque C standard et des autres bibliothèques complémentaires), ainsi que la résolution des références inter-fichiers. Cette étape est appelée *édition de liens* (« linking » en anglais). Le résultat de l'édition de liens est le *fichier image*, qui pourra être chargé en mémoire par le système d'exploitation. Les fichiers exécutables et les bibliothèques dynamiques sont des exemples de fichiers image.

Figure 6-1. Processus de génération des binaires



Toutes ces opérations peuvent être régroupées en une seule étape par les outils utilisés. Ainsi, les compilateurs appellent généralement le préprocesseur et l'assembleur automatiquement, et réalisent parfois même l'édition de liens eux-mêmes. Toutefois, il reste généralement possible, à l'aide d'options spécifiques à chaque outil de développement, de décomposer les différentes étapes et d'obtenir les fichiers intermédiaires.

En raison du nombre de fichiers important et des dépendances qui peuvent exister entre eux, le processus de génération d'un programme prend très vite une certaine ampleur. Les deux problèmes les plus courants sont de déterminer l'ordre dans lequel les fichiers et les bibliothèques doivent être compilés, ainsi que les dépendances entre fichiers sources et les fichiers produits afin de pouvoir regénérer

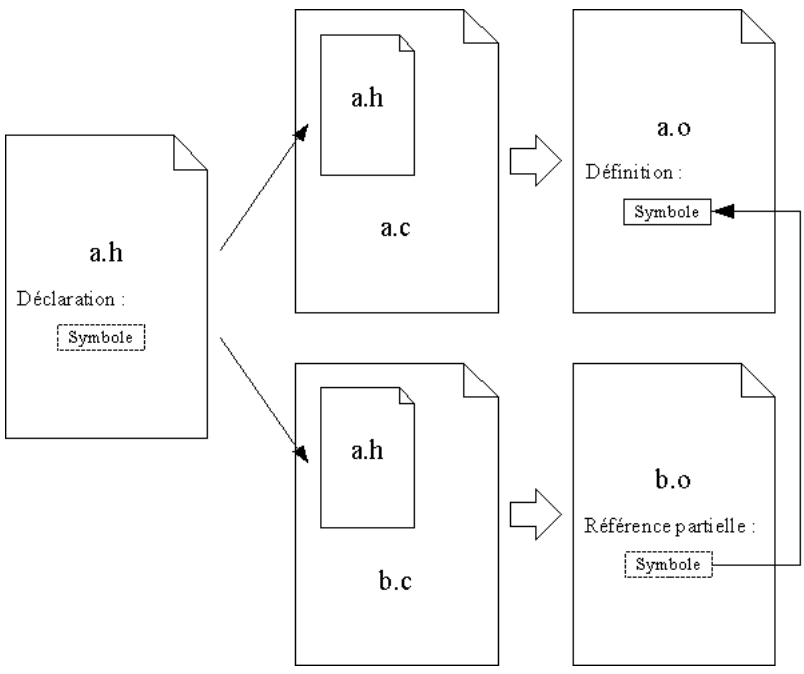
correctement les fichiers images après une modification des sources. Tous ces problèmes peuvent être résolus à l'aide d'un programme appelé **make**. Le principe de **make** est toujours le même, même si aucune norme n'a été définie en ce qui le concerne. **make** lit un fichier (le fichier (« `makefile` »), dans lequel se trouvent toutes les opérations nécessaires pour compiler un programme. Puis, il les exécute si c'est nécessaire. Par exemple, un fichier qui a déjà été compilé et qui n'a pas été modifié depuis ne sera pas recompilé. C'est plus rapide. **make** se base sur les dates de dernière modification des fichiers pour savoir s'ils ont été modifiés (il compare les dates des fichiers sources et des fichiers produits). La date des fichiers est gérée par le système d'exploitation : il est donc important que l'ordinateur soit à l'heure.

6.3. Compilation séparée en C/C++

La compilation séparée en C/C++ se fait au niveau du fichier. Il existe trois grands types de fichiers sources en C/C++ :

- les fichiers d'en-tête, qui contiennent toutes les déclarations communes à plusieurs fichiers sources. Ce sont les fichiers d'en-têtes qui, en séparant la déclaration de la définition des symboles du programme, permettent de découper l'ensemble des sources en fichiers compilables séparément ;
- les fichiers C, qui contiennent les définitions des symboles en langage C ;
- les fichiers C++, qui contiennent les définitions des symboles en langage C++.

Les programmes modulaires C/C++ auront donc typiquement la structure suivante :



Note : Il faudra bien faire la distinction entre les fichiers sources compilés séparément et les fichiers inclus par le préprocesseur. Ces derniers sont en effet compilés avec les fichiers dans

lesquels ils sont inclus. Il n'est donc pas recommandé d'inclure des définitions de symboles dans les fichiers d'en-tête, car ces symboles risquent d'apparaître dans plusieurs fichiers objets après la compilation. Cela provoque généralement une erreur à l'édition de liens, parce que l'éditeur de liens ne peut pas déterminer quelle définition prendre parmi celles qui se trouvent dans les différents fichiers objets.

6.4. Syntaxe des outils de compilation

Il existe évidemment un grand nombre de compilateurs C/C++ pour chaque plateforme. Ils ne sont malheureusement pas compatibles au niveau de la ligne de commande. Le même problème apparaît pour les éditeurs de liens (« linker » en anglais) et pour **make**. Cependant, quelques principes généraux peuvent être établis. Dans la suite, je supposerai que le nom du compilateur est « **cc** », que celui du préprocesseur est « **cpp** », celui de l'éditeur de liens est « **ld** » et que celui de make est « **make** ».

En général, les différentes étapes de la compilation et de l'édition de liens sont regroupées au niveau du compilateur, ce qui permet de faire les phases de traitement du préprocesseur, de compilation et d'édition de liens en une seule commande. Les lignes de commandes des compilateurs sont donc souvent compliquées et très peu portable. En revanche, la syntaxe de `make` est un peu plus portable.

6.4.1. Syntaxe des compilateurs

Le compilateur demande en général les noms des fichiers sources à compiler et les noms des fichiers objets à utiliser lors de la phase d'édition de liens. Lorsque l'on spécifie un fichier source, le compilateur utilisera le fichier objet qu'il aura créé pour ce fichier source en plus des fichiers objets donnés dans la ligne de commande. Le compilateur peut aussi accepter en ligne de commande le chemin de recherche des bibliothèques du langage et des fichiers d'en-tête. Enfin, différentes options d'optimisation sont disponibles (mais très peu portables). La syntaxe (simplifiée) des compilateurs est souvent la suivante :

```
cc [fichier.o [...]] [[-c] fichier.c [...]] [-o exécutable]
   [-Lchemin_bibliothèques] [-lbibliothèque [...]] [-Ichemin_include]
```

`fichier.c` est le nom du fichier à compiler. Si l'option `-c` le précède, le fichier sera compilé, mais l'éditeur de liens ne sera pas appelé. Si cette option n'est pas présente, l'éditeur de liens est appelé, et le programme exécutable formé est enregistré dans le fichier `a.out`. Pour donner un autre nom à ce programme, il faut utiliser l'option `-o`, suivie du nom de l'exécutable. Il est possible de donner le nom des fichiers objets déjà compilés (« `fichier.o` ») pour que l'éditeur de liens les lie avec le programme compilé.

L'option `-L` permet d'indiquer le chemin du répertoire des bibliothèques de fonctions prédéfinies. Ce répertoire sera ajouté à la liste des répertoires indiqués dans la variable d'environnement LIBRARY_PATH. L'option `-l` demande au compilateur d'utiliser la bibliothèque spécifiée, si elle ne fait pas partie des bibliothèques utilisées par défaut. De même, l'option `-I` permet de donner le chemin d'accès au répertoire des fichiers à inclure (lors de l'utilisation du préprocesseur). Les chemins ajoutés avec cette option viennent s'ajouter aux chemins indiqués dans les variables d'environnement C_INCLUDE_PATH et CPLUS_INCLUDE_PATH pour les programmes compilés respectivement en C et en C++.

L'ordre des paramètres sur la ligne de commande est significatif. La ligne de commande est exécutée de gauche à droite.

Exemple 6-1. Compilation d'un fichier et édition de liens

```
cc -c fichier1.c
cc fichier1.o programme.cc -o lancez_moi
```

Dans cet exemple, le fichier C `fichier1.c` est compilé en `fichier1.o`, puis le fichier C++ `programme.cc` est compilé et lié au `fichier1.o` pour former l'exécutable `lancez_moi`.

6.4.2. Syntaxe de make

La syntaxe de make est très simple :

```
make
```

En revanche, la syntaxe du fichier `makefile` est un peu plus compliquée et peu portable. Cependant, les fonctionnalités de base sont gérées de la même manière par la plupart des programme **make**.

Le fichier `makefile` est constitué d'une série de lignes d'information et de lignes de commande (de l'interpréteur de commandes UNIX ou DOS). Les commandes doivent toujours être précédées d'un caractère de tabulation horizontale.

Les lignes d'information donnent des renseignements sur les dépendances des fichiers (en particulier, les fichiers objets qui doivent être utilisés pour créer l'exécutable). Les lignes d'information permettent donc à **make** d'identifier les fichiers sources à compiler afin de générer l'exécutable. Les lignes de commande indiquent comment effectuer cette compilation (et éventuellement d'autres tâches).

La syntaxe des lignes d'information est la suivante :

```
nom:dépendance
```

où `nom` est le nom de la cible (généralement, il s'agit du nom du fichier destination), et `dépendance` est la liste des noms des fichiers dont dépend cette cible, séparés par des espaces. La syntaxe des lignes de commande utilisée est celle de l'interpréteur du système hôte. Enfin, les commentaires dans un fichier makefile se font avec le signe dièse (`#`).

Exemple 6-2. Fichier makefile sans dépendances

```
# Compilation du fichier fichier1.c :
 cc - c fichier1.c

# Compilation du programme principal :
 cc -o Lancez_moi fichier1.o programme.c
```

Exemple 6-3. Fichier makefile avec dépendances

```
# Indique les dépendances :
Lancez_moi: fichier1.o programme.o

# Indique comment compiler le programme :
# (le symbole $@ représente le nom de la cible, ici, Lancez_moi)
 cc -o $@ fichier1.o programme.o

#compile les dépendances :
```

```
fichier1.o: fichier1.c
 cc -c fichier1.c

programme.o: programme1.c
 cc -c programme.c
```

6.5. Problèmes syntaxiques relatifs à la compilation séparée

Pour que le compilateur puisse compiler les fichiers séparément, il faut que vous respectiez les conditions suivantes :

- chaque type ou variable utilisé doit être déclaré ;
- toute fonction non déclarée doit renvoyer un entier (en C seulement, en C++, l'utilisation d'une fonction non déclarée génère une erreur).

Ces conditions ont des répercussions sur la syntaxe des programmes. Elles seront vues dans les paragraphes suivants.

6.5.1. Déclaration des types

Les types doivent toujours être définis avant toute utilisation dans un fichier source. Par exemple, il est interdit d'utiliser une structure client sans l'avoir définie avant sa première utilisation. Toutefois, il est possible d'utiliser un pointeur sur un type de donnée sans l'avoir complètement défini. Une simple déclaration du type de base du pointeur suffit en effet dans ce cas là. De même, un simple `class MaClasse` suffit en C++ pour déclarer une classe sans la définir complètement.

6.5.2. Déclaration des variables

Les variables qui sont définies dans un autre fichier doivent être déclarées avant leur première utilisation. Pour cela, on les spécifie comme étant des variables externes, avec le mot clé `extern` :

```
extern int i;   /* i est un entier qui est déclaré et
                   créé dans un autre fichier.
                   Ici, il est simplement déclaré.
                */
```

Inversement, si une variable ne doit pas être accédée par un autre module, il faut déclarer cette variable statique. Ainsi, même si un autre fichier utilise le mot clé `extern`, il ne pourra pas y accéder.

6.5.3. Déclaration des fonctions

Lorsqu'une fonction se trouve définie dans un autre fichier, il est nécessaire de la déclarer. Pour cela, il suffit de donner sa déclaration (le mot clé `extern` est également utilisable, mais facultatif dans ce cas) :

```
int factorielle(int);
/*
   factorielle est une fonction attendant comme paramètre
   un entier et renvoyant une valeur entière.
   Elle est définie dans un autre fichier.
 */
```

Les fonctions `inline` doivent impérativement être définies dans les fichiers où elles sont utilisées, puisqu'en théorie, elles sont recopiées dans les fonctions qui les utilisent. Cela implique de placer leur définition dans les fichiers d'en-tête `.h` ou `.hpp`. Comme le code des fonctions `inline` est normalement inclus dans le code des fonctions qui les utilisent, les fichiers d'en-tête contenant du code `inline` peuvent être compilés séparément sans que ces fonctions ne soient définies plusieurs fois. Par conséquent, l'éditeur de liens ne générera pas d'erreur (alors qu'il l'aurait fait si on avait placé le code d'une fonction non `inline` dans un fichier d'en-tête inclus dans plusieurs fichiers sources `.c` ou `.cpp`). Certains programmeurs considèrent qu'il n'est pas bon de placer des définitions de fonctions dans des fichiers d'en-tête, il placent donc toutes leurs fonctions `inline` dans des fichiers portant l'extension `.inl`. Ces fichiers sont ensuite inclus soit dans les fichiers d'en-tête `.h`, soit dans les fichiers `.c` ou `.cpp` qui utilisent les fonctions `inline`.

6.5.4. Directives d'édition de liens

Le langage C++ donne la possibilité d'appeler des fonctions et d'utiliser des variables qui proviennent d'un module écrit dans un autre langage. Pour permettre cela, il dispose de directives permettant d'indiquer comment l'édition de liens doit être faite. La syntaxe permettant de réaliser cela utilise le mot clé `extern`, avec le nom du langage entre guillemets. Cette directive d'édition de liens doit précéder les déclarations de variables et de données concernées. Si plusieurs variables ou fonctions utilisent la même directive, elles peuvent être regroupées dans un bloc délimité par des accolades, avec la directive d'édition de liens placée juste avant ce bloc. La syntaxe est donc la suivante :

```
extern "langage" [déclaration | {
déclaration
[...]
}]
```

Cependant, les seuls langages qu'une implémentation doit obligatoirement supporter sont les langages « C » et « C++ ». Pour les autres langages, aucune norme n'est définie et les directives d'édition de liens sont dépendantes de l'implémentation.

Exemple 6-4. Déclarations utilisables en C et en C++

```
#ifdef __cplusplus
extern "C"
{
#endif
```

```
    extern int EntierC;
    int FonctionC(void);
#ifdef __cplusplus
}
#endif
```

Dans l'exemple précédent, la compilation conditionnelle est utilisée pour n'utiliser la directive d'édition de liens que si le code est compilé en C++. Si c'est le cas, la variable `EntierC` et la fonction `FonctionC` sont déclarées au compilateur C++ comme étant des objets provenant d'un module C.

Chapitre 7. C++ : la couche objet

La couche objet constitue sans doute la plus grande innovation du C++ par rapport au C. Le but de la programmation objet est de permettre une abstraction entre l'implémentation des modules et leur utilisation, apportant ainsi un plus grand confort dans la programmation. Elle s'intègre donc parfaitement dans le cadre de la modularité. Enfin, l'encapsulation des données permet une meilleure protection et donc une plus grande fiabilité des programmes.

7.1. Généralités

Théoriquement, il y a une nette distinction entre les données et les opérations qui leur sont appliquées. En tout cas, les données et le code ne se mélangent pas dans la mémoire de l'ordinateur, sauf cas très particuliers (autoprogrammation, alias pour le chargement des programmes ou des overlays, débogueurs, virus).

Cependant, l'analyse des problèmes à traiter se présente d'une manière plus naturelle si l'on considère les données avec leurs propriétés. Les données constituent les variables, et les propriétés les opérations qu'on peut leur appliquer. De ce point de vue, les données et le code sont logiquement inséparables, même s'ils sont placés en différents endroits de la mémoire de l'ordinateur.

Ces considérations conduisent à la notion d'objet. Un *objet* est un ensemble de données sur lesquelles des procédures peuvent être appliquées. Ces procédures ou fonctions applicables aux données sont appelées *méthodes*. La programmation d'un objet se fait donc en indiquant les données de l'objet et en définissant les procédures qui peuvent lui être appliquées.

Il se peut qu'il y ait plusieurs objets identiques, dont les données ont bien entendu des valeurs différentes, mais qui utilisent le même jeu de méthodes. On dit que ces différents objets appartiennent à la même *classe* d'objets. Une classe constitue donc une sorte de type, et les objets de cette classe en sont des *instances*. La classe définit donc la structure des données, alors appelées *champs* ou *variables d'instances*, que les objets correspondants auront, ainsi que les méthodes de l'objet. À chaque instanciation, une allocation de mémoire est faite pour les données du nouvel objet créé. L'initialisation de l'objet nouvellement créé est faite par une méthode spéciale, le *constructeur*. Lorsque l'objet est détruit, une autre méthode est appelée : le *destructeur*. L'utilisateur peut définir ses propres constructeurs et destructeurs d'objets si nécessaire.

Comme seules les valeurs des données des différents objets d'une classe diffèrent, les méthodes sont mises en commun pour tous les objets d'une même classe (c'est-à-dire que les méthodes ne sont pas recopiées). Pour que les méthodes appelées pour un objet sachent sur quelles données elles doivent travailler, un pointeur sur l'objet contenant ces données leur est passé en paramètre. Ce mécanisme est complètement transparent pour le programmeur en C++.

Nous voyons donc que non seulement la programmation orientée objet est plus logique, mais elle est également plus efficace (les méthodes sont mises en commun, les données sont séparées).

Enfin, les données des objets peuvent être *protégées* : c'est-à-dire que seules les méthodes de l'objet peuvent y accéder. Ce n'est pas une obligation, mais cela accroît la fiabilité des programmes. Si une erreur se produit, seules les méthodes de l'objet doivent être vérifiées. De plus, les méthodes constituent ainsi une interface entre les données de l'objet et l'utilisateur de l'objet (un autre programmeur). Cet utilisateur n'a donc pas à savoir comment les données sont gérées dans l'objet, il ne doit utiliser que les méthodes. Les avantages sont immédiats : il ne risque pas de faire des erreurs de programmation en modifiant les données lui-même, l'objet est réutilisable dans un autre programme parce qu'il a une interface standardisée, et on peut modifier l'implémentation interne de l'objet sans avoir à refaire tout le programme, pourvu que les méthodes gardent le même nom, les mêmes paramètres et la même

sémantique. Cette notion de protection des données et de masquage de l'implémentation interne aux utilisateurs de l'objet constitue ce que l'on appelle l'*encapsulation*. Les avantages de l'encapsulation seront souvent mis en valeur dans la suite au travers d'exemples.

Nous allons entrer maintenant dans le vif du sujet. Cela permettra de comprendre ces généralités.

7.2. Extension de la notion de type du C

Il faut avant tout savoir que la couche objet n'est pas un simple ajout au langage C, c'est une véritable extension. En effet, les notions qu'elle a apportées ont été intégrées au C à tel point que le typage des données de C a fusionné avec la notion de classe. Ainsi, les types prédéfinis char, int, double, etc. représentent à présent l'ensemble des propriétés des variables ayant ce type. Ces propriétés constituent la classe de ces variables, et elles sont accessibles par les opérateurs. Par exemple, l'addition est une opération pouvant porter sur des entiers (entre autres) qui renvoie un objet de la classe entier. Par conséquent, les types de base se manipuleront exactement comme des objets. Du point de vue du C++, les utiliser revient déjà à faire de la programmation orientée objet.

De même, le programmeur peut, à l'aide de la notion de classe d'objets, définir de nouveaux types. Ces types comprennent la structure des données représentées par ces types et les opérations qui peuvent leur être appliquées. En fait, le C++ assimile complètement les classes avec les types, et la définition d'un nouveau type se fait donc en définissant la classe des variables de ce type.

7.3. Déclaration de classes en C++

Afin de permettre la définition des méthodes qui peuvent être appliquées aux structures des classes C++, la syntaxe des structures C a été étendue (et simplifiée). Il est à présent possible de définir complètement des méthodes dans la définition de la structure. Cependant il est préférable de la reporter et de ne laisser que leur déclaration dans la structure. En effet, cela accroît la lisibilité et permet de masquer l'implémentation de la classe à ses utilisateurs en ne leur montrant que sa déclaration dans un fichier d'en-tête. Ils ne peuvent donc ni la voir, ni la modifier (en revanche, ils peuvent toujours voir la structure de données utilisée par son implémentation).

La syntaxe est la suivante :

```
struct Nom
{
    [type champs;
    [type champs;
    [...]]]

    [méthode;
    [méthode;
    [...]]]
};
```

où `Nom` est le nom de la classe. Elle peut contenir divers champs de divers types.

Les méthodes peuvent être des définitions de fonctions, ou seulement leurs déclarations. Si on ne donne que leurs déclarations, on devra les définir plus loin. Pour cela, il faudra spécifier la classe à laquelle elles appartiennent avec la syntaxe suivante :

```
type classe::nom(paramètres)
{
```

```
    /* Définition de la méthode. */
}
```

La syntaxe est donc identique à la définition d'une fonction normale, à la différence près que leur nom est précédé du nom de la classe à laquelle elles appartiennent et de deux deux-points (`::`). Cet opérateur `::` est appelé l'*opérateur de résolution de portée*. Il permet, d'une manière générale, de spécifier le bloc auquel l'objet qui le suit appartient. Ainsi, le fait de précéder le nom de la méthode par le nom de la classe permet au compilateur de savoir de quelle classe cette méthode fait partie. Rien n'interdit, en effet, d'avoir des méthodes de même signature, pourvu qu'elles soient dans des classes différentes.

Exemple 7-1. Déclaration de méthodes de classe

```
struct Entier
{
    int i;                // Donnée membre de type entier.

    // Fonction définie à l'intérieur de la classe :
    int lit_i(void)
    {
        return i;
    }

    // Fonction définie à l'extérieur de la classe :
    void ecrit_i(int valeur);
};

void Entier::ecrit_i(int valeur)
{
    i=valeur;
    return ;
}
```

Note : Si la liste des paramètres de la définition de la fonction contient des initialisations supplémentaires à celles qui ont été spécifiées dans la déclaration de la fonction, les deux jeux d'initialisations sont fusionnées et utilisées dans le fichier où la définition de la fonction est placée. Si les initialisations sont redondantes ou contradictoires, le compilateur génère une erreur.

Note : L'opérateur de résolution de portée permet aussi de spécifier le bloc d'instructions d'un objet qui n'appartient à aucune classe. Pour cela, on ne mettra aucun nom avant l'opérateur de résolution de portée. Ainsi, pour accéder à une fonction globale à l'intérieur d'une classe contenant une fonction de même signature, on fera précéder le nom de la fonction globale de cet opérateur.

Exemple 7-2. Opérateur de résolution de portée

```
int valeur(void)         // Fonction globale.
{
    return 0;
}

struct A
{
```

```
    int i;

    void fixe(int a)
    {
        i=a;
        return;
    }

    int valeur(void)       // Même signature que la fonction globale.
    {
        return i;
    }

    int global_valeur(void)
    {
        return ::valeur(); // Accède à la fonction globale.
    }
};
```

De même, l'opérateur de résolution de portée permettra d'accéder à une variable globale lorsqu'une autre variable homonyme aura été définie dans le bloc en cours. Par exemple :

```
#include <stdlib.h>

int i=1;                // Première variable de portée globale

int main(void)
{
    if (test())
    {
       int i=3;         // Variable homonyme de portée locale.
       int j=2*::i;     // j vaut à présent 2, et non pas 6.
       /* Suite ... */
    }

    /* Suite ... */

    return EXIT_SUCCESS;
}
```

Les champs d'une classe peuvent être accédés comme des variables normales dans les méthodes de cette classe.

Exemple 7-3. Utilisation des champs d'une classe dans une de ses méthodes

```
struct client
{
    char Nom[21], Prenom[21];    // Définit le client.
    unsigned int Date_Entree;    // Date d'entrée du client
                                 // dans la base de données.
    int Solde;

    bool dans_le_rouge(void)
    {
        return (Solde<0);
    }
```

```
    bool bon_client(void)        // Le bon client est
                                 // un ancien client.
    {
        return (Date_Entree<1993); // Date limite : 1993.
    }
};
```

Dans cet exemple, le client est défini par certaines données. Plusieurs méthodes sont définies dans la classe même.

L'instanciation d'un objet se fait comme celle d'une simple variable :

```
classe objet;
```

Par exemple, si on a une base de données devant contenir 100 clients, on peut faire :

```
client clientele[100];  /* Instancie 100 clients. */
```

On remarquera qu'il est à présent inutile d'utiliser le mot clé `struct` pour déclarer une variable, contrairement à ce que la syntaxe du C exigeait.

L'accès aux méthodes de la classe se fait comme pour accéder aux champs des structures. On donne le nom de l'objet et le nom du champ ou de la méthode, séparés par un point. Par exemple :

```
/* Relance de tous les mauvais payeurs. */
int i;
for (i=0; i<100; ++i)
    if (clientele[i].dans_le_rouge()) relance(clientele[i]);
```

Lorsque les fonctions membres d'une classe sont définies dans la déclaration de cette classe, le compilateur les implémente en `inline` (à moins qu'elles ne soient récursives ou qu'il existe un pointeur sur elles).

Si les méthodes ne sont pas définies dans la classe, la déclaration de la classe sera mise dans un fichier d'en-tête, et la définition des méthodes sera reportée dans un fichier C++, qui sera compilé et lié aux autres fichiers utilisant la classe client. Bien entendu, il est toujours possible de déclarer les fonctions membres comme étant des fonctions `inline` même lorsqu'elles sont définies en dehors de la déclaration de la classe. Pour cela, il faut utiliser le mot clé `inline`, et placer le code de ces fonctions dans le fichier d'en-tête ou dans un fichier `.inl`.

Sans fonctions inline, notre exemple devient :

Fichier client.h :

```
struct client
{
    char Nom[21], Prenom[21];
    unsigned int Date_Entree;
    int Solde;

    bool dans_le_rouge(void);
    bool bon_client(void);
```

```
};

/*
Attention à ne pas oublier le ; à la fin de la classe dans un
fichier .h ! L'erreur apparaîtrait dans tous les fichiers ayant
une ligne #include "client.h" , parce que la compilation a lieu
après l'appel au préprocesseur.
*/
```

Fichier client.cc :

```
/* Inclut la déclaration de la classe : */
#include "client.h"

/* Définit les méthodes de la classe : */

bool client::dans_le_rouge(void)
{
    return (Solde<0);
}

bool client::bon_client(void)
{
    return (Date_Entree<1993);
}
```

7.4. Encapsulation des données

Les divers champs d'une structure sont accessibles en n'importe quel endroit du programme. Une opération telle que celle-ci est donc faisable :

```
clientele[0].Solde = 25000;
```

Le solde d'un client peut donc être modifié sans passer par une méthode dont ce serait le but. Elle pourrait par exemple vérifier que l'on n'affecte pas un solde supérieur au solde maximal autorisé par le programme (la borne supérieure des valeurs des entiers signés). Par exemple, si les entiers sont codés sur 16 bits, cette borne maximum est 32767. Un programme qui ferait :

```
clientele[0].Solde = 32800;
```

obtiendrait donc un solde de -12 (valeur en nombre signé du nombre non signé 32800), alors qu'il espérerait obtenir un solde positif !

Il est possible d'empêcher l'accès des champs ou de certaines méthodes à toute fonction autre que celles de la classe. Cette opération s'appelle l'encapsulation. Pour la réaliser, il faut utiliser les mots clés suivants :

- `public` : les accès sont libres ;
- `private` : les accès sont autorisés dans les fonctions de la classe seulement ;

- `protected` : les accès sont autorisés dans les fonctions de la classe et de ses descendantes (voir la section suivante) seulement. Le mot clé `protected` n'est utilisé que dans le cadre de l'héritage des classes. La section suivante détaillera ce point.

Pour changer les droits d'accès des champs et des méthodes d'une classe, il faut faire précéder ceux-ci du mot clé indiquant les droits d'accès suivi de deux points (':'). Par exemple, pour protéger les données relatives au client, on changera simplement la déclaration de la classe en :

```
struct client
{
private:   // Données privées :

    char Nom[21], Prenom[21];
    unsigned int Date_Entree;
    int Solde;
    // Il n'y a pas de méthode privée.

public:    // Les données et les méthodes publiques :

    // Il n'y a pas de donnée publique.
    bool dans_le_rouge(void);
    bool bon_client(void)
};
```

Outre la vérification de la validité des opérations, l'encapsulation a comme intérêt fondamental de définir une interface stable pour la classe au niveau des méthodes et données membres publiques et protégées. L'implémentation de cette interface, réalisée en privé, peut être modifiée à loisir sans pour autant perturber les utilisateurs de cette classe, tant que cette interface n'est pas elle-même modifiée.

Par défaut, les classes construites avec `struct` ont tous leurs membres publics. Il est possible de déclarer une classe dont tous les éléments sont par défaut privés. Pour cela, il suffit d'utiliser le mot clé `class` à la place du mot clé `struct`.

Exemple 7-4. Utilisation du mot clé class

```
class client
{
    // private est à présent inutile.

    char Nom[21], Prenom[21];
    unsigned int Date_Entree;
    int Solde;

public:    // Les données et les méthodes publiques.

    bool dans_le_rouge(void);
    bool bon_client(void);
};
```

Enfin, il existe un dernier type de classe, que je me contenterai de mentionner : les classes *union*. Elles se déclarent comme les classes `struct` et `class`, mais avec le mot clé `union`. Les données sont, comme pour les unions du C, situées toutes au même emplacement, ce qui fait qu'écrire dans l'une

d'entre elle provoque la destruction des autres. Les unions sont très souvent utilisées en programmation système, lorsqu'un polymorphisme physique des données est nécessaire (c'est-à-dire lorsqu'elles doivent être interprétées de différentes façons selon le contexte).

> **Note :** Les classes de type `union` ne peuvent pas avoir de méthodes virtuelles et de membres statiques. Elles ne peuvent pas avoir de classes de base, ni servir de classe de base. Enfin, les unions ne peuvent pas contenir des références, ni des objets dont la classe a un constructeur non trivial, un constructeur de copie non trivial ou un destructeur non trivial. Pour toutes ces notions, voir la suite du chapitre.
>
> Les classes définies au sein d'une autre classe n'ont pas de droits particuliers sur leur classe hôte. De même, la classe hôte n'a pas plus de droits spécifiques sur les membres de ses sous-classes. Notez que nombre de compilateurs ne respectent pas scrupuleusement ces règles, et donnent parfois des droits aux classes hôtes. Pour autant, le paragraphe 11.8 de la norme C++ est très clair à ce sujet. Il vous faudra donc déclarer amie la classe hôte dans les classes qui sont définies en son sein si vous voulez accéder à leurs membres librement. La manière de procéder sera décrite dans la Section 7.7.2.

7.5. Héritage

L'*héritage* permet de donner à une classe toutes les caractéristiques d'une ou de plusieurs autres classes. Les classes dont elle hérite sont appelées *classes mères*, *classes de base* ou *classes antécédentes*. La classe elle-même est appelée *classe fille*, *classe dérivée* ou *classe descendante*.

Les *propriétés héritées* sont les champs et les méthodes des classes de base.

Pour faire un héritage en C++, il faut faire suivre le nom de la classe fille par la liste des classes mères dans la déclaration avec les restrictions d'accès aux données, chaque élément étant séparé des autres par une virgule. La syntaxe (donnée pour `class`, identique pour `struct`) est la suivante :

```
class Classe_mere1
{
    /* Contenu de la classe mère 1. */
};

[class Classe_mere2
{
    /* Contenu de la classe mère 2. */
};]

[...]

class Classe_fille : public|protected|private Classe_mere1
[, public|protected|private Classe_mere2 [...]]
{
    /* Définition de la classe fille. */
};
```

Dans cette syntaxe, `Classe_fille` hérite de la `Classe_mere1`, et des `Classe_mere2`, etc. si elles sont présentes.

La signification des mots clés `private`, `protected` et `public` dans l'héritage est récapitulée dans le tableau suivant :

Tableau 7-1. Droits d'accès sur les membres hérités

Accès aux données		mot clé utilisé pour l'héritage		
	public	**protected**	**private**	
mot clé utilisé	`public`	`public`	`protected`	`private`
pour les champs	`protected`	`protected`	`protected`	`private`
et les méthodes	`private`	`interdit`	`interdit`	`interdit`

Ainsi, les données publiques d'une classe mère deviennent soit publiques, soit protégées, soit privées selon que la classe fille hérite en public, protégé ou en privé. Les données privées de la classe mère sont toujours inaccessibles, et les données protégées deviennent soit protégées, soit privées.

Il est possible d'omettre les mots clés `public`, `protected` et `private` dans la syntaxe de l'héritage. Le compilateur utilise un type d'héritage par défaut dans ce cas. Les classes de type `struct` utilisent l'héritage `public` par défaut et les classes de type `class` utilisent le mot clé `private` par défaut.

Exemple 7-5. Héritage public, privé et protégé

```
class Emplacement
{
protected:
    int x, y;              // Données ne pouvant être accédées
                           // que par les classes filles.

public:
    void Change(int, int); // Méthode toujours accessible.
};

void Emplacement::Change(int i, int j)
{
    x = i;
    y = j;
    return;
}

class Point : public Emplacement
{
protected:
    unsigned int couleur;  // Donnée accessible
                           // aux classes filles.

public:
    void SetColor(unsigned int);
};

void Point::SetColor(unsigned int NewColor)
{
    couleur = NewColor;    // Définit la couleur.
    return;
}
```

Si une classe Cercle doit hériter de deux classes mères, par exemple Emplacement et Forme, sa déclaration aura la forme suivante :

```
class Cercle : public Emplacement, public Forme
{
    /*
       Définition de la classe Cercle. Cette classe hérite
       des données publiques et protégées des classes Emplacement
       et Forme.
     */
};
```

Les membres des classes de base auxquels les classes dérivées ont accès peuvent être redéclarés avec de nouveaux droits d'accès par celles-ci. Par exemple, une donnée membre publique héritée publiquement peut être protégée unitairement. Inversement, une donnée membre protégée peut être rendue publique dans la classe dérivée. Cela se fait via une simple redéclaration à l'aide du mot clé `using`, avec des droits d'accès différents. Ce mot clé s'emploie comme suit :

```
using Base::membre;
```

où `membre` est le nom du membre de la classe de base que l'on veut redéclarer. Nous verrons plus en détail les diverses utilisation de ce mot clé dans la Section 10.2.

Il est possible de redéfinir les fonctions et les données des classes de base dans une classe dérivée. Par exemple, si une classe B dérive de la classe A, et que toutes deux contiennent une donnée `d`, les instances de la classe B utiliseront la donnée `d` de la classe B et les instances de la classe A utiliseront la donnée `d` de la classe A. Cependant, les objets de classe B contiendront également un sous-objet, lui-même instance de la classe de base A. Par conséquent, ils contiendront la donnée `d` de la classe A, mais cette dernière sera cachée par la donnée `d` de la classe la plus dérivée, à savoir la classe B.

Ce mécanisme est général : quand une classe dérivée redéfinit un membre d'une classe de base, ce membre est caché et on ne peut plus accéder directement qu'au membre redéfini (celui de la classe dérivée). Cependant, il est possible d'accéder aux données cachées si l'on connaît leur classe, pour cela, il faut nommer le membre complètement à l'aide de l'opérateur de résolution de portée (`::`). Le nom complet d'un membre est constitué du nom de sa classe suivi de l'opérateur de résolution de portée, suivis du nom du membre :

```
classe::membre
```

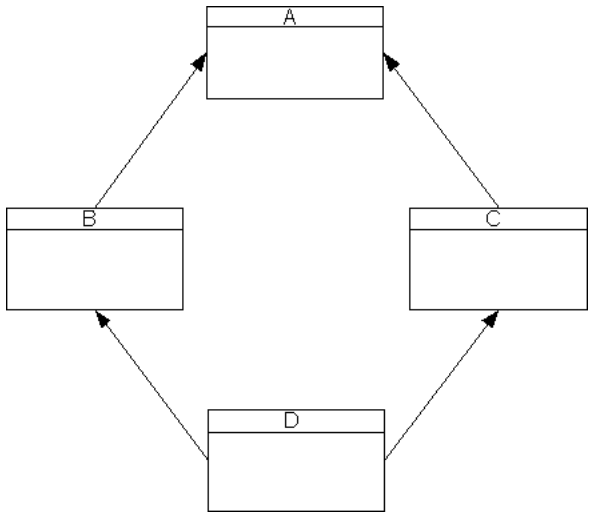
Exemple 7-6. Opérateur de résolution de portée et membre de classes de base

```
#include <stdlib.h>

struct Base
{
    int i;
};

struct Derivee : public Base
{
    int i;
    int LitBase(void);
};

int Derivee::LitBase(void)
{
```

```
    return Base::i; // Renvoie la valeur i de la classe de base.
}

int main(void)
{
    Derivee D;
    D.i=1;          // Accède à l'entier i de la classe Derivee.
    D.Base::i=2;    // Accède à l'entier i de la classe Base.
    return EXIT_SUCCESS;
}
```

7.6. Classes virtuelles

Supposons à présent qu'une classe D hérite de deux classes mères, les classes B et C. Supposons également que ces deux classes héritent d'une classe mère commune appelée classe A. On a l'arbre « généalogique » suivant :



On sait que B et C héritent des données et des méthodes publiques et protégées de A. De même, D hérite des données de B et C, et par leur intermédiaire des données de A. Il se pose donc le problème suivant : quelles sont les données que l'on doit utiliser quand on référence les champs de A ? Celles de B ou celles de C ? On peut accéder aux deux sous-objets de classe A en spécifiant le chemin à suivre dans l'arbre généalogique à l'aide de l'opérateur de résolution de portée. Cependant, cela n'est ni pratique ni efficace, et en général, on s'attend à ce qu'une seule copie de A apparaisse dans D.

Le problème est résolu en déclarant *virtuelle* la classe de base commune dans la spécification de l'héritage pour les classes filles. Les données de la classe de base ne seront alors plus dupliquées. Pour déclarer une classe mère comme une classe virtuelle, il faut faire précéder son nom du mot clé `virtual` dans l'héritage des classes filles.

Exemple 7-7. Classes virtuelles

```
class A
{
```

```
protected:
    int Donnee;         // La donnée de la classe de base.
};

// Héritage de la classe A, virtuelle :
class B : virtual public A
{
protected:
    int Valeur_B;       // Autre donnée que "Donnee" (héritée).
};

// A est toujours virtuelle :
class C : virtual public A
{
protected:
    int valeur_C;       // Autre donnée
                        // ("Donnee" est acquise par héritage).
};

class D : public B, public C   // Ici, Donnee n'est pas dupliqué.
{
    /* Définition de la classe D. */
};
```

Note : Normalement, l'héritage est réalisé par le compilateur par aggrégation de la structure de données des classes de base dans la structure de données de la classe dérivée. Pour les classes virtuelles, ce n'est en général pas le cas, puisque le compilateur doit assurer l'unicité des données héritées de ces classes, même en cas d'héritage multiple. Par conséquent, certaines restrictions d'usage s'appliquent sur les classes virtuelles.

Premièrement, il est impossible de transtyper directement un pointeur sur un objet d'une classe de base virtuelle en un pointeur sur un objet d'une de ses classes dérivées. Il faut impérativement utiliser l'opérateur de transtypage dynamique `dynamic_cast`. Cet opérateur sera décrit dans le Chapitre 9.

Deuxièmement, chaque classe dérivée directement ou indirectement d'une classe virtuelle doit en appeler le constructeur explicitement dans son constructeur si celui-ci prend des paramètres. En effet, elle ne peut pas se fier au fait qu'une autre de ses classes de base, elle-même dérivée de la classe de base virtuelle, appelle un constructeur spécifique, car il est possible que plusieurs classes de base cherchent à initialiser différemment chacune un objet commun hérité de la classe virtuelle. Pour reprendre l'exemple donné ci-dessus, si les classes B et C appellaient toutes les deux un constructeur non trivial de la classe virtuelle A, et que la classe D appellait elle-même les constructeurs de B et C, le sous-objet hérité de A serait construit plusieurs fois. Pour éviter cela, le compilateur ignore purement et simplement les appels au constructeur des classes de bases virtuelles dans les classes de base dérivées. Il faut donc systématiquement le spécifier, à chaque niveau de la hiérarchie de classe. La notion de constructeur sera vue dans la Section 7.8.

7.7. Fonctions et classes amies

Il est parfois nécessaire d'avoir des fonctions qui ont un accès illimité aux champs d'une classe. En général, l'emploi de telles fonctions traduit un manque d'analyse dans la hiérarchie des classes, mais pas toujours. Elles restent donc nécessaires malgré tout.

De telles fonctions sont appelées des *fonctions amies*. Pour qu'une fonction soit amie d'une classe, il faut qu'elle soit déclarée dans la classe avec le mot clé `friend`.

Il est également possible de faire une classe amie d'une autre classe, mais dans ce cas, cette classe devrait peut-être être une classe fille. L'utilisation des classes amies peut traduire un défaut de conception.

7.7.1. Fonctions amies

Les fonctions amies se déclarent en faisant précéder la déclaration classique de la fonction du mot clé `friend` à l'intérieur de la déclaration de la classe cible. Les fonctions amies ne sont pas des méthodes de la classe cependant (cela n'aurait pas de sens puisque les méthodes ont déjà accès aux membres de la classe).

Exemple 7-8. Fonctions amies

```
class A
{
    int a;                          // Une donnée privée.
    friend void ecrit_a(int i);     // Une fonction amie.
};

A essai;

void ecrit_a(int i)
{
    essai.a=i;          // Initialise a.
    return;
}
```

Il est possible de déclarer amie une fonction d'une autre classe, en précisant son nom complet à l'aide de l'opérateur de résolution de portée.

7.7.2. Classes amies

Pour rendre toutes les méthodes d'une classe amies d'une autre classe, il suffit de déclarer la classe complète comme étant amie. Pour cela, il faut encore une fois utiliser le mot clé `friend` avant la déclaration de la classe, à l'intérieur de la classe cible. Cette fois encore, la classe amie déclarée ne sera pas une sous-classe de la classe cible, mais bien une classe de portée globale.

> **Note :** Le fait, pour une classe, d'appartenir à une autre classe lui donne le droit d'accéder aux membres de sa classe hôte. Il n'est donc pas nécessaire de déclarer amies d'une classe les classes définies au sein de celle-ci. Remarquez que cette règle a été récemment modifiée dans la norme C++, et que la plupart des compilateurs refuseront aux classes incluses d'accéder aux membres non publics de leur conteneur.

Exemple 7-9. Classe amie

```
#include <stdlib.h>
#include <stdio.h>
```

```
class Hote
{
    friend class Amie;  // Toutes les méthodes de Amie sont amies.

    int i;              // Donnée privée de la classe Hote.

public:
    Hote(void)
    {
        i=0;
        return ;
    }
};

Hote h;

class Amie
{
public:
    void print_hote(void)
    {
        printf("%d\n", h.i); // Accède à la donnée privée de h.
        return ;
    }
};

int main(void)
{
    Amie a;
    a.print_hote();
    return EXIT_SUCCESS;
}
```

On remarquera plusieurs choses importantes. Premièrement, l'amitié n'est pas transitive. Cela signifie que les amis des amis ne sont pas des amis. Une classe A amie d'une classe B, elle-même amie d'une classe C, n'est pas amie de la classe C par défaut. Il faut la déclarer amie explicitement si on désire qu'elle le soit. Deuxièmement, les amis ne sont pas hérités. Ainsi, si une classe A est amie d'une classe B et que la classe C est une classe fille de la classe B, alors A n'est pas amie de la classe C par défaut. Encore une fois, il faut la déclarer amie explicitement. Ces remarques s'appliquent également aux fonctions amies (une fonction amie d'une classe A amie d'une classe B n'est pas amie de la classe B, ni des classes dérivées de A).

7.8. Constructeurs et destructeurs

Le *constructeur* et le *destructeur* sont deux méthodes particulières qui sont appelées respectivement à la création et à la destruction d'un objet. Toute classe a un constructeur et un destructeur par défaut, fournis par le compilateur. Ces constructeurs et destructeurs appellent les constructeurs par défaut et les destructeurs des classes de base et des données membres de la classe, mais en dehors de cela, ils ne font absolument rien. Il est donc souvent nécessaire de les redéfinir afin de gérer certaines actions qui doivent avoir lieu lors de la création d'un objet et de leur destruction. Par exemple, si l'objet doit contenir des variables allouées dynamiquement, il faut leur réserver de la mémoire à la création de l'objet ou au moins mettre les pointeurs correspondants à `NULL`. À la destruction de l'objet, il convient

de restituer la mémoire allouée, s'il en a été alloué. On peut trouver bien d'autres situations où une phase d'initialisation et une phase de terminaison sont nécessaires.

Dès qu'un constructeur ou un destructeur a été défini par l'utilisateur, le compilateur ne définit plus automatiquement le constructeur ou le destructeur par défaut correspondant. En particulier, si l'utilisateur définit un constructeur prenant des paramètres, il ne sera plus possible de construire un objet simplement, sans fournir les paramètres à ce constructeur, à moins bien entendu de définir également un constructeur qui ne prenne pas de paramètres.

7.8.1. Définition des constructeurs et des destructeurs

Le constructeur se définit comme une méthode normale. Cependant, pour que le compilateur puisse la reconnaître en tant que constructeur, les deux conditions suivantes doivent être vérifiées :

- elle doit porter le même nom que la classe ;
- elle ne doit avoir aucun type, *pas même le type void*.

Le destructeur doit également respecter ces règles. Pour le différencier du constructeur, son nom sera toujours précédé du signe tilde ('~').

Un constructeur est appelé automatiquement lors de l'instanciation de l'objet. Le destructeur est appelé automatiquement lors de sa destruction. Cette destruction a lieu lors de la sortie du bloc de portée courante pour les objets de classe de stockage `auto`. Pour les objets alloués dynamiquement, le constructeur et le destructeur sont appelés automatiquement par les expressions qui utilisent les opérateurs `new`, `new[]`, `delete` et `delete[]`. C'est pour cela qu'il est recommandé de les utiliser à la place des fonctions `malloc` et `free` du C pour créer dynamiquement des objets. De plus, il ne faut pas utiliser `delete` ou `delete[]` sur des pointeurs de type void, car il n'existe pas d'objets de type void. Le compilateur ne peut donc pas déterminer quel est le destructeur à appeler avec ce type de pointeur.

Le constructeur est appelé après l'allocation de la mémoire de l'objet et le destructeur est appelé avant la libération de cette mémoire. La gestion de l'allocation dynamique de mémoire avec les classes est ainsi simplifiée. Dans le cas des tableaux, l'ordre de construction est celui des adresses croissantes, et l'ordre de destruction est celui des adresses décroissantes. C'est dans cet ordre que les constructeurs et destructeurs de chaque élément du tableau sont appelés.

Les constructeurs pourront avoir des paramètres. Ils peuvent donc être surchargés, mais pas les destructeurs. Cela est dû a fait qu'en général on connaît le contexte dans lequel un objet est créé, mais qu'on ne peut pas connaître le contexte dans lequel il est détruit : il ne peut donc y avoir qu'un seul destructeur. Les constructeurs qui ne prennent pas de paramètre ou dont tous les paramètres ont une valeur par défaut, remplacent automatiquement les constructeurs par défaut définis par le compilateur lorsqu'il n'y a aucun constructeur dans les classes. Cela signifie que ce sont ces constructeurs qui seront appelés automatiquement par les constructeurs par défaut des classes dérivées.

Exemple 7-10. Constructeurs et destructeurs

```
class chaine    // Implémente une chaîne de caractères.
{
    char * s;   // Le pointeur sur la chaîne de caractères.

public:
    chaine(void);           // Le constructeur par défaut.
```

```
    chaine(unsigned int);   // Le constructeur. Il n'a pas de type.
    ~chaine(void);          // Le destructeur.
};

chaine::chaine(void)
{
    s=NULL;                 // La chaîne est initialisée avec
    return ;                // le pointeur nul.
}

chaine::chaine(unsigned int Taille)
{
    s = new char[Taille+1]; // Alloue de la mémoire pour la chaîne.
    s[0]='\0';              // Initialise la chaîne à "".
    return;
}

chaine::~chaine(void)
{
    if (s!=NULL) delete[] s; // Restitue la mémoire utilisée si
                             // nécessaire.
    return;
}
```

Pour passer les paramètres au constructeur, on donne la liste des paramètres entre parenthèses juste après le nom de l'objet lors de son instanciation :

```
chaine s1;        // Instancie une chaîne de caractères
                  // non initialisée.
chaine s2(200);   // Instancie une chaîne de caractères
                  // de 200 caractères.
```

Les constructeurs devront parfois effectuer des tâches plus compliquées que celles données dans cet exemple. En général, ils peuvent faire toutes les opérations faisables dans une méthode normale, sauf utiliser les données non initialisées bien entendu. En particulier, les données des sous-objets d'un objet ne sont pas initialisées tant que les constructeurs des classes de base ne sont pas appelés. C'est pour cela qu'il faut toujours appeler les constructeurs des classes de base avant d'exécuter le constructeur de la classe en cours d'instanciation. Si les constructeurs des classes de base ne sont pas appelés explicitement, le compilateur appellera, par défaut, les constructeurs des classes mères qui ne prennent pas de paramètre ou dont tous les paramètres ont une valeur par défaut (et, si aucun constructeur n'est défini dans les classe mères, il appellera les constructeurs par défaut de ces classes).

Comment appeler les constructeurs et les destructeurs des classes mères lors de l'instanciation et de la destruction d'une classe dérivée ? Le compilateur ne peut en effet pas savoir quel constructeur il faut appeler parmi les différents constructeurs surchargés potentiellement présents... Pour appeler un autre constructeur d'une classe de base que le constructeur ne prenant pas de paramètre, il faut spécifier explicitement ce constructeur avec ses paramètres après le nom du constructeur de la classe fille, en les séparant de deux points (':').

En revanche, il est inutile de préciser le destructeur à appeler, puisque celui-ci est unique. Le programmeur ne doit donc pas appeler lui-même les destructeurs des classes mères, le langage s'en charge.

Exemple 7-11. Appel du constructeur des classes de base

```
/* Déclaration de la classe mère. */

class Mere
{
    int m_i;
public:
    Mere(int);
    ~Mere(void);
};

/* Définition du constructeur de la classe mère. */

Mere::Mere(int i)
{
    m_i=i;
    printf("Exécution du constructeur de la classe mère.\n");
    return;
}

/* Définition du destructeur de la classe mère. */

Mere::~Mere(void)
{
    printf("Exécution du destructeur de la classe mère.\n");
    return;
}

/* Déclaration de la classe fille. */

class Fille : public Mere
{
public:
    Fille(void);
    ~Fille(void);
};

/* Définition du constructeur de la classe fille
   avec appel du constructeur de la classe mère. */

Fille::Fille(void) : Mere(2)
{
    printf("Exécution du constructeur de la classe fille.\n");
    return;
}

/* Définition du destructeur de la classe fille
   avec appel automatique du destructeur de la classe mère. */

Fille::~Fille(void)
{
    printf("Exécution du destructeur de la classe fille.\n");
    return;
}
```

Lors de l'instanciation d'un objet de la classe fille, le programme affichera dans l'ordre les messages suivants :

```
Exécution du constructeur de la classe mère.
Exécution du constructeur de la classe fille.
```

et lors de la destruction de l'objet :

```
Exécution du destructeur de la classe fille.
Exécution du destructeur de la classe mère.
```

Si l'on n'avait pas précisé que le constructeur à appeler pour la classe Mere était le constructeur prenant un entier en paramètre, le compilateur aurait essayé d'appeler le constructeur par défaut de cette classe. Or, ce constructeur n'étant plus généré automatiquement par le compilateur (à cause de la définition d'un constructeur prenant un paramètre), il y aurait eu une erreur de compilation.

Il est possible d'appeler plusieurs constructeurs si la classe dérive de plusieurs classes de base. Pour cela, il suffit de lister les constructeurs un à un, en séparant leurs appels par des virgules. On notera cependant que l'ordre dans lequel les constructeurs sont appelés n'est pas forcément l'ordre dans lequel ils sont listés dans la définition du constructeur de la classe fille. En effet, le C++ appelle toujours les constructeurs dans l'ordre d'apparition de leurs classes dans la liste des classes de base de la classe dérivée.

> **Note :** Afin d'éviter l'utilisation des données non initialisées de l'objet le plus dérivé dans une hiérarchie pendant la construction de ses sous-objets par l'intermédiaire des fonctions virtuelles, le mécanisme des fonctions virtuelles est désactivé dans les constructeurs (voyez la Section 7.13 pour plus de détails sur les fonctions virtuelles). Ce problème survient parce que pendant l'exécution des constructeurs des classes de base, l'objet de la classe en cours d'instanciation n'a pas encore été initialisé, et malgré cela, une fonction virtuelle aurait pu utiliser une donnée de cet objet.
>
> Une fonction virtuelle peut donc toujours être appelée dans un constructeur, mais la fonction effectivement appelée est celle de la classe du sous-objet en cours de construction : pas celle de la classe de l'objet complet. Ainsi, si une classe A hérite d'une classe B et qu'elles ont toutes les deux une fonction virtuelle `f`, l'appel de `f` dans le constructeur de B utilisera la fonction `f` de B, pas celle de A (même si l'objet que l'on instancie est de classe A).

La syntaxe utilisée pour appeler les constructeurs des classes de base peut également être utilisée pour initialiser les données membres de la classe. En particulier, cette syntaxe est obligatoire pour les données membres constantes et pour les références, car le C++ ne permet pas l'affectation d'une valeur à des variables de ce type. Encore une fois, l'ordre d'appel des constructeurs des données membres ainsi initialisées n'est pas forcément l'ordre dans lequel ils sont listés dans le constructeur de la classe. En effet, le C++ utilise cette fois l'ordre de déclaration de chaque donnée membre.

Exemple 7-12. Initialisation de données membres constantes

```
class tableau
{
    const int m_iTailleMax;
    const int *m_pDonnees;
public:
    tableau(int iTailleMax);
    ~tableau();
};
```

```
tableau::tableau(int iTailleMax) :
    m_iTailleMax(iTailleMax)    // Initialise la donnée membre constante.
{
    // Allocation d'un tableau de m_iTailleMax entrées :
    m_pDonnees = new int[m_iTailleMax];
}

tableau::~tableau()
{
    // Destruction des données :
    delete[] m_pDonnees;
}
```

Note : Les constructeurs des classes de base virtuelles prenant des paramètres doivent être appelés par chaque classe qui en dérive, que cette dérivation soit directe ou indirecte. En effet, les classes de base virtuelles subissent un traitement particulier qui assure l'unicité de leurs données dans toutes leurs classes dérivées. Les classes dérivées ne peuvent donc pas se reposer sur leurs classes de base pour appeler le constructeur des classes virtuelles, car il peut y avoir plusieurs classes de bases qui dérivent d'une même classe virtuelle, et cela supposerait que le constructeur de cette dernière classe serait appelé plusieurs fois, éventuellement avec des valeurs de paramètres différentes. Chaque classe doit donc prendre en charge la construction des sous-objets des classes de base virtuelles dont il hérite dans ce cas.

7.8.2. Constructeurs de copie

Il faudra parfois créer un constructeur de copie. Le but de ce type de constructeur est d'initialiser un objet lors de son instanciation à partir d'un autre objet. Toute classe dispose d'un constructeur de copie par défaut généré automatiquement par le compilateur, dont le seul but est de recopier les champs de l'objet à recopier un à un dans les champs de l'objet à instancier. Toutefois, ce constructeur par défaut ne suffira pas toujours, et le programmeur devra parfois en fournir un explicitement.

Ce sera notamment le cas lorsque certaines données des objets auront été allouées dynamiquement. Une copie brutale des champs d'un objet dans un autre ne ferait que recopier les pointeurs, pas les données pointées. Ainsi, la modification de ces données pour un objet entraînerait la modification des données de l'autre objet, ce qui ne serait sans doute pas l'effet désiré.

La définition des constructeurs de copie se fait comme celle des constructeurs normaux. Le nom doit être celui de la classe, et il ne doit y avoir aucun type. Dans la liste des paramètres cependant, il devra toujours y avoir une référence sur l'objet à copier.

Pour la classe chaine définie ci-dessus, il faut un constructeur de copie. Celui-ci peut être déclaré de la façon suivante :

```
chaine(const chaine &Source);
```

où `Source` est l'objet à copier.

Si l'on rajoute la donnée membre *`Taille`* dans la déclaration de la classe, la définition de ce constructeur peut être :

```
chaine::chaine(const chaine &Source)
{
```

```
    int i = 0;                    // Compteur de caractères.
    Taille = Source.Taille;
    s = new char[Taille + 1];     // Effectue l'allocation.
    strcpy(s, Source.s);          // Recopie la chaîne de caractères source.
    return;
}
```

Le constructeur de copie est appelé dans toute instanciation avec initialisation, comme celles qui suivent :

```
chaine s2(s1);
chaine s2 = s1;
```

Dans les deux exemples, c'est le constructeur de copie qui est appelé. En particulier, à la deuxième ligne, le constructeur normal n'est pas appelé et aucune affectation entre objets n'a lieu.

> **Note :** Le fait de définir un constructeur de copie pour une classe signifie généralement que le constructeur de copie, le destructeur et l'opérateur d'affectation fournis par défaut par le compilateur ne conviennent pas pour cette classe. Par conséquent, ces méthodes devront systématiquement être redéfinies toutes les trois dès que l'une d'entre elle le sera. Cette règle, que l'on appelle la *règle des trois*, vous permettra d'éviter des bogues facilement. Vous trouverez de plus amples détails sur la manière de redéfinir l'opérateur d'affectation dans la Section 7.11.3.
>
> Les constructeurs de copie et les destructeurs ne devront pas avoir d'autres effets que ceux dictés par leur sémantique. En effet, les compilateurs peuvent parfaitement effectuer des optimisations et ne pas appeler ces méthodes dans certaines situations (généralement, pour éviter des copies d'objets temporaires inutiles). Par conséquent, les constructeurs et les destructeurs ne devront généralement pas avoir d'effets de bord.

7.8.3. Utilisation des constructeurs dans les transtypages

Les constructeurs sont utilisés dans les conversions de type dans lesquelles le type cible est celui de la classe du constructeur. Ces conversions peuvent être soit implicites (dans une expression), soit explicite (à l'aide d'un transtypage). Par défaut, les conversions implicites sont légales, pourvu qu'il existe un constructeur dont le premier paramètre a le même type que l'objet source. Par exemple, la classe Entier suivante :

```
class Entier
{
    int i;
public:
    Entier(int j)
    {
        i=j;
        return ;
    }
};
```

dispose d'un constructeur de transtypage pour les entiers. Les expressions suivantes :

```
int j=2;
Entier e1, e2=j;
e1=j;
```

sont donc légales, la valeur entière située à la droite de l'expression étant convertie implicitement en un objet du type de la classe Entier.

Si, pour une raison quelconque, ce comportement n'est pas souhaitable, on peut forcer le compilateur à n'accepter que les conversions explicites (à l'aide de transtypage). Pour cela, il suffit de placer le mot clé `explicit` avant la déclaration du constructeur. Par exemple, le constructeur de la classe chaine vue ci-dessus prenant un entier en paramètre risque d'être utilisé dans des conversions implicites. Or ce constructeur ne permet pas de construire une chaîne de caractères à partir d'un entier, et ne doit donc pas être utilisé dans les opérations de transtypage. Ce constructeur doit donc être déclaré `explicit` :

```
class chaine
{
    size_t Taille;
    char * s;

public:
    chaine(void);
    // Ce constructeur permet de préciser la taille de la chaîne
 // à sa création :
    explicit chaine(unsigned int);
    ~chaine(void);
};
```

Avec cette déclaration, l'expression suivante :

```
int j=2;
chaine s = j;
```

n'est plus valide, alors qu'elle l'était lorsque le constructeur n'était pas déclaré `explicit`.

> **Note :** On prendra garde au fait que le mot clé `explicit` n'empêche l'utilisation du constructeur dans les opérations de transtypage que dans les conversions implicites. Si le transtypage est explicitement demandé, le constructeur sera malgré tout utilisé. Ainsi, le code suivant sera accepté :
>
> ```
> int j=2;
> chaine s = (chaine) j;
> ```
>
> Bien entendu, cela n'a pas beaucoup de signification et ne devrait jamais être effectué.

7.9. Pointeur this

Nous allons à présent voir comment les fonctions membres, qui appartiennent à la classe, peuvent accéder aux données d'un objet, qui est une instance de cette classe. Cela est indispensable pour bien comprendre les paragraphes suivants.

À chaque appel d'une fonction membre, le compilateur passe implicitement un pointeur sur les données de l'objet en paramètre. Ce paramètre est le premier paramètre de la fonction. Ce mécanisme est complètement invisible au programmeur, et nous ne nous attarderons pas dessus.

En revanche, il faut savoir que le pointeur sur l'objet est accessible à l'intérieur de la fonction membre. Il porte le nom « `this` ». Par conséquent, `*this` représente l'objet lui-même. Nous verrons une utilisation de `this` dans le paragraphe suivant (surcharge des opérateurs).

`this` est un pointeur constant, c'est-à-dire qu'on ne peut pas le modifier (il est donc impossible de faire des opérations arithmétiques dessus). Cela est tout à fait normal, puisque le faire reviendrait à sortir de l'objet en cours (celui pour lequel la méthode en cours d'exécution travaille).

Il est possible de transformer ce pointeur constant en un pointeur constant sur des données constantes pour chaque fonction membre. Le pointeur ne peut toujours pas être modifié, et les données de l'objet ne peuvent pas être modifiées non plus. L'objet est donc considéré par la fonction membre concernée comme un objet constant. Cela revient à dire que la fonction membre s'interdit la modification des données de l'objet. On parvient à ce résultat en ajoutant le mot clé `const` à la suite de l'en-tête de la fonction membre. Par exemple :

```
class Entier
{
    int i;
public:
    int lit(void) const;
};

int Entier::lit(void) const
{
    return i;
}
```

Dans la fonction membre `lit`, il est impossible de modifier l'objet. On ne peut donc accéder qu'en lecture seule à `i`. Nous verrons une application de cette possibilité dans la Section 7.15.

Il est à noter qu'une méthode qui n'est pas déclarée comme étant `const` modifie a priori les données de l'objet sur lequel elle travaille. Donc, si elle est appelée sur un objet déclaré `const`, une erreur de compilation se produit. Ce comportement est normal. On devra donc toujours déclarer `const` une méthode qui ne modifie pas réellement l'objet, afin de laisser à l'utilisateur le choix de déclarer `const` ou non les objets de sa classe.

> **Note :** Le mot clé `const` n'intervient pas dans la signature des fonctions en général lorsqu'il s'applique aux paramètres (tout paramètre déclaré `const` perd sa qualification dans la signature). En revanche, il intervient dans la signature d'une fonction membre quand il s'applique à cette fonction (ou, plus précisément, à l'objet pointé par `this`). Il est donc possible de déclarer deux fonctions membres acceptant les mêmes paramètres, dont une seule est `const`. Lors de l'appel, la détermination de la fonction à utiliser dépendra de la nature de l'objet sur lequel elle doit s'appliquer. Si l'objet est `const`, la méthode appelée sera celle qui est `const`.

7.10. Données et fonctions membres statiques

Nous allons voir dans ce paragraphe l'emploi du mot clé `static` dans les classes. Ce mot clé intervient pour caractériser les données membres statiques des classes, les fonctions membres statiques des classes, et les données statiques des fonctions membres.

7.10.1. Données membres statiques

Une classe peut contenir des données membres statiques. Ces données sont soit des données membres propres à la classe, soit des données locales statiques des fonctions membres de la classe. Dans tous les cas, elles appartiennent à la classe, et non pas aux objets de cette classe. Elles sont donc communes à tous ces objets.

Il est impossible d'initialiser les données d'une classe dans le constructeur de la classe, car le constructeur n'initialise que les données des nouveaux objets. Les données statiques ne sont pas spécifiques à un objet particulier et ne peuvent donc pas être initialisées dans le constructeur. En fait, leur initialisation doit se faire lors de leur définition, en dehors de la déclaration de la classe. Pour préciser la classe à laquelle les données ainsi définies appartiennent, on devra utiliser l'opérateur de résolution de portée (`::`).

Exemple 7-13. Donnée membre statique

```
class test
{
    static int i;       // Déclaration dans la classe.
    ...
};

int test::i=3;         // Initialisation en dehors de la classe.
```

La variable `test::i` sera partagée par tous les objets de classe test, et sa valeur initiale est `3`.

> **Note :** La définition des données membres statiques suit les mêmes règles que la définition des variables globales. Autrement dit, elles se comportent comme des variables déclarées externes. Elles sont donc accessibles dans tous les fichiers du programme (pourvu, bien entendu, qu'elles soient déclarées en zone publique dans la classe). De même, elles ne doivent être définies qu'une seule fois dans tout le programme. Il ne faut donc pas les définir dans un fichier d'en-tête qui peut être inclus plusieurs fois dans des fichiers sources, même si l'on protège ce fichier d'en-tête contre les inclusions multiples.

Les variables statiques des fonctions membres doivent être initialisées à l'intérieur des fonctions membres. Elles appartiennent également à la classe, et non pas aux objets. De plus, leur portée est réduite à celle du bloc dans lequel elles ont été déclarées. Ainsi, le code suivant :

```
#include <stdlib.h>
#include <stdio.h>

class test
{
public:
    int n(void);
};
```

```
int test::n(void)
{
    static int compte=0;
    return compte++;
}

int main(void)
{
    test objet1, objet2;
    printf("%d ", objet1.n());   // Affiche 0
    printf("%d\n", objet2.n());  // Affiche 1
    return EXIT_SUCCESS;
}
```

affichera `0` et `1`, parce que la variable statique `compte` est la même pour les deux objets.

7.10.2. Fonctions membres statiques

Les classes peuvent également contenir des fonctions membres statiques. Cela peut surprendre à première vue, puisque les fonctions membres appartiennent déjà à la classe, c'est-à-dire à tous les objets. En fait, cela signifie que ces fonctions membres ne recevront pas le pointeur sur l'objet `this`, comme c'est le cas pour les autres fonctions membres. Par conséquent, elles ne pourront accéder qu'aux données statiques de l'objet.

Exemple 7-14. Fonction membre statique

```
class Entier
{
    int i;
    static int j;
public:
    static int get_value(void);
};

int Entier::j=0;

int Entier::get_value(void)
{
    j=1;         // Légal.
    return i;    // ERREUR ! get_value ne peut pas accéder à i.
}
```

La fonction `get_value` de l'exemple ci-dessus ne peut pas accéder à la donnée membre non statique *`i`*, parce qu'elle ne travaille sur aucun objet. Son champ d'action est uniquement la classe Entier. En revanche, elle peut modifier la variable statique *`j`*, puisque celle-ci appartient à la classe Entier et non aux objets de cette classe.

L'appel des fonctions membre statiques se fait exactement comme celui des fonctions membres non statiques, en spécifiant l'identificateur d'un des objets de la classe et le nom de la fonction membre, séparés par un point. Cependant, comme les fonctions membres ne travaillent pas sur les objets des classes mais plutôt sur les classes elles-mêmes, la présence de l'objet lors de l'appel est facultatif. On peut donc se contenter d'appeler une fonction statique en qualifiant son nom du nom de la classe à laquelle elle appartient à l'aide de l'opérateur de résolution de portée.

Exemple 7-15. Appel de fonction membre statique

```
#include <stdlib.h>

class Entier
{
    static int i;
public:
    static int get_value(void);
};

int Entier::i=3;

int Entier::get_value(void)
{
    return i;
}

int main(void)
{
    // Appelle la fonction statique get_value :
    int resultat=Entier::get_value();
    return EXIT_SUCCESS;
}
```

Les fonctions membres statiques sont souvent utilisées afin de regrouper un certain nombre de fonctionnalités en rapport avec leur classe. Ainsi, elles sont facilement localisable et les risques de conflits de noms entre deux fonctions membres homonymes sont réduits. Nous verrons également dans le Chapitre 10 comment éviter les conflits de noms globaux dans le cadre des espaces de nommage.

7.11. Surcharge des opérateurs

On a vu précédemment que les opérateurs ne se différencient des fonctions que syntaxiquement, pas logiquement. D'ailleurs, le compilateur traite un appel à un opérateur comme un appel à une fonction. Le C++ permet donc de surcharger les opérateurs pour les classes définies par l'utilisateur, en utilisant une syntaxe particulière calquée sur la syntaxe utilisée pour définir des fonctions membres normales. En fait, il est même possible de surcharger les opérateurs du langage pour les classes de l'utilisateur en dehors de la définition de ces classes. Le C++ dispose donc de deux méthodes différentes pour surcharger les opérateurs.

Les seuls opérateurs qui ne peuvent pas être surchargés sont les suivants :

```
::
.
.*
?:
sizeof
typeid
static_cast
dynamic_cast
const_cast
reinterpret_cast
```

Tous les autres opérateurs sont surchargeables. Leur surcharge ne pose généralement pas de problème et peut être réalisée soit dans la classe des objets sur lesquels ils s'appliquent, soit à l'extérieur de

cette classe. Cependant, un certain nombre d'entre eux demandent des explications complémentaires, que l'on donnera à la fin de cette section.

> **Note :** On prendra garde aux problèmes de performances lors de la surcharge des opérateurs. Si la facilité d'écriture des expressions utilisant des classes est grandement simplifiée grâce à la possibilité de surcharger les opérateurs pour ces classes, les performances du programme peuvent en être gravement affectées. En effet, l'utilisation inconsidérée des opérateurs peut conduire à un grand nombre de copies des objets, copies que l'on pourrait éviter en écrivant le programme classiquement. Par exemple, la plupart des opérateurs renvoient un objet du type de la classe sur laquelle ils travaillent. Ces objets sont souvent créés localement dans la fonction de l'opérateur (c'est-à-dire qu'ils sont de portée `auto`). Par conséquent, ces objets sont temporaires et sont détruits à la sortie de la fonction de l'opérateur. Cela impose donc au compilateur d'en faire une copie dans la valeur de retour de la fonction avant d'en sortir. Cette copie sera elle-même détruite par le compilateur une fois qu'elle aura été utilisée par l'instruction qui a appelé la fonction. Si le résultat doit être affecté à un objet de l'appelant, une deuxième copie inutile est réalisée par rapport au cas où l'opérateur aurait travaillé directement dans la variable résultat. Si les bons compilateurs sont capables d'éviter ces copies, cela reste l'exception et il vaut mieux être averti à l'avance plutôt que de devoir réécrire tout son programme a posteriori pour des problèmes de performances.

Nous allons à présent voir dans les sections suivantes les deux syntaxes permettant de surcharger les opérateurs pour les types de l'utilisateur, ainsi que les règles spécifiques à certains opérateurs particuliers.

7.11.1. Surcharge des opérateurs internes

Une première méthode pour surcharger les opérateurs consiste à les considérer comme des méthodes normales de la classe sur laquelle ils s'appliquent. Le nom de ces méthodes est donné par le mot clé `operator`, suivi de l'opérateur à surcharger. Le type de la fonction de l'opérateur est le type du résultat donné par l'opération, et les paramètres, donnés entre parenthèses, sont les opérandes. Les opérateurs de ce type sont appelés opérateurs internes, parce qu'ils sont déclarés à l'intérieur de la classe.

Voici la syntaxe :

```
type operatorOp(paramètres)
```

l'écriture

```
A Op B
```

se traduisant par :

```
A.operatorOp(B)
```

Avec cette syntaxe, le premier opérande est toujours l'objet auquel cette fonction s'applique. Cette manière de surcharger les opérateurs est donc particulièrement bien adaptée pour les opérateurs qui modifient l'objet sur lequel ils travaillent, comme par exemple les opérateurs `=`, `+=`, `++`, etc. Les paramètres de la fonction opérateur sont alors le deuxième opérande et les suivants.

Les opérateurs définis en interne devront souvent renvoyer l'objet sur lequel ils travaillent (ce n'est pas une nécessité cependant). Cela est faisable grâce au pointeur `this`.

Par exemple, la classe suivante implémente les nombres complexes avec quelques-unes de leurs opérations de base.

Exemple 7-16. Surcharge des opérateurs internes

```
class complexe
{
    double m_x, m_y;  // Les parties réelles et imaginaires.
public:
    // Constructeurs et opérateur de copie :
    complexe(double x=0, double y=0);
    complexe(const complexe &);
    complexe &operator=(const complexe &);

    // Fonctions permettant de lire les parties réelles
    // et imaginaires :
    double re(void) const;
    double im(void) const;

    // Les opérateurs de base:
    complexe &operator+=(const complexe &);
    complexe &operator-=(const complexe &);
    complexe &operator*=(const complexe &);
    complexe &operator/=(const complexe &);
};

complexe::complexe(double x, double y)
{
    m_x = x;
    m_y = y;
    return ;
}

complexe::complexe(const complexe &source)
{
    m_x = source.m_x;
    m_y = source.m_y;
    return ;
}

complexe &complexe::operator=(const complexe &source)
{
    m_x = source.m_x;
    m_y = source.m_y;
    return *this;
}

double complexe::re() const
{
    return m_x;
}

double complexe::im() const
{
    return m_y;
}
```

```
complexe &complexe::operator+=(const complexe &c)
{
    m_x += c.m_x;
    m_y += c.m_y;
    return *this;
}

complexe &complexe::operator-=(const complexe &c)
{
    m_x -= c.m_x;
    m_y -= c.m_y;
    return *this;
}

complexe &complexe::operator*=(const complexe &c)
{
    double temp = m_x*c.m_x -m_y*c.m_y;
    m_y = m_x*c.m_y + m_y*c.m_x;
    m_x = temp;
    return *this;
}

complexe &complexe::operator/=(const complexe &c)
{
    double norm = c.m_x*c.m_x + c.m_y*c.m_y;
    double temp = (m_x*c.m_x + m_y*c.m_y) / norm;
    m_y = (-m_x*c.m_y + m_y*c.m_x) / norm;
    m_x = temp;
    return *this;
}
```

> **Note :** La bibliothèque standard C++ fournit une classe traitant les nombres complexes de manière complète, la classe complex. Cette classe n'est donc donnée ici qu'à titre d'exemple et ne devra évidemment pas être utilisée. La définition des nombres complexes et de leur principales propriétés sera donnée dans la Section 14.3.1, où la classe complex sera décrite.

Les opérateurs d'affectation fournissent un exemple d'utilisation du pointeur `this`. Ces opérateurs renvoient en effet systématiquement l'objet sur lequel ils travaillent, afin de permettre des affectations multiples. Les opérateurs de ce type devront donc tous se terminer par :

```
return *this;
```

7.11.2. Surcharge des opérateurs externes

Une deuxième possibilité nous est offerte par le langage pour surcharger les opérateurs. La définition de l'opérateur ne se fait plus dans la classe qui l'utilise, mais en dehors de celle-ci, par surcharge d'un opérateur de l'espace de nommage global. Il s'agit donc d'opérateurs externes cette fois.

La surcharge des opérateurs externes se fait donc exactement comme on surcharge les fonctions normales. Dans ce cas, tous les opérandes de l'opérateur devront être passés en paramètres : il n'y aura pas de paramètre implicite (le pointeur `this` n'est pas passé en paramètre).

La syntaxe est la suivante :

```
type operatorOp(opérandes)
```

où `opérandes` est la liste complète des opérandes.

L'avantage de cette syntaxe est que l'opérateur est réellement symétrique, contrairement à ce qui se passe pour les opérateurs définis à l'intérieur de la classe. Ainsi, si l'utilisation de cet opérateur nécessite un transtypage sur l'un des opérandes, il n'est pas nécessaire que cet opérande soit obligatoirement le deuxième. Donc si la classe dispose de constructeurs permettant de convertir un type de donnée en son prope type, ce type de donnée peut être utilisé avec tous les opérateurs de la classe.

Par exemple, les opérateurs d'addition, de soustraction, de multiplication et de division de la classe complexe peuvent être implémentés comme dans l'exemple suivant.

Exemple 7-17. Surcharge d'opérateurs externes

```
class complexe
{
    friend complexe operator+(const complexe &, const complexe &);
    friend complexe operator-(const complexe &, const complexe &);
    friend complexe operator*(const complexe &, const complexe &);
    friend complexe operator/(const complexe &, const complexe &);

    double m_x, m_y;  // Les parties réelles et imaginaires.
public:
    // Constructeurs et opérateur de copie :
    complexe(double x=0, double y=0);
    complexe(const complexe &);
    complexe &operator=(const complexe &);

    // Fonctions permettant de lire les parties réelles
    // et imaginaires :
    double re(void) const;
    double im(void) const;

    // Les opérateurs de base:
    complexe &operator+=(const complexe &);
    complexe &operator-=(const complexe &);
    complexe &operator*=(const complexe &);
    complexe &operator/=(const complexe &);
};

// Les opérateurs de base ont été éludés ici :
...

complexe operator+(const complexe &c1, const complexe &c2)
{
    complexe result = c1;
    return result += c2;
}

complexe operator-(const complexe &c1, const complexe &c2)
{
    complexe result = c1;
    return result -= c2;
}
```

```
complexe operator*(const complexe &c1, const complexe &c2)
{
    complexe result = c1;
    return result *= c2;
}

complexe operator/(const complexe &c1, const complexe &c2)
{
    complexe result = c1;
    return result /= c2;
}
```

Avec ces définitions, il est parfaitement possible d'effectuer la multiplication d'un objet de type complexe avec une valeur de type double. En effet, cette valeur sera automatiquement convertie en complexe grâce au constructeur de la classe complexe, qui sera utilisé ici comme constructeur de transtypage. Une fois cette conversion effectuée, l'opérateur adéquat est appliqué.

On constatera que les opérateurs externes doivent être déclarés comme étant des fonctions amies de la classe sur laquelle ils travaillent, faute de quoi ils ne pourraient pas manipuler les données membres de leurs opérandes.

> **Note :** Certains compilateurs peuvent supprimer la création des variables temporaires lorsque celles-ci sont utilisées en tant que valeur de retour des fonctions. Cela permet d'améliorer grandement l'efficacité des programmes, en supprimant toutes les copies d'objets inutiles. Cependant ces compilateurs sont relativement rares et peuvent exiger une syntaxe particulière pour effectuer cette optimisation. Généralement, les compilateurs C++ actuels suppriment la création de variable temporaire dans les retours de fonctions si la valeur de retour est construite dans l'instruction `return` elle-même. Par exemple, l'opérateur d'addition peut être optimisé ainsi :
>
> ```
> complexe operator+(const complexe &c1, const complexe &c2)
> {
>     return complexe(c1.m_x + c2.m_x, c1.m_y + c2.m_y);
> }
> ```
>
> Cette écriture n'est cependant pas toujours utilisable, et l'optimisation n'est pas garantie.

La syntaxe des opérateurs externes permet également d'implémenter les opérateurs pour lesquels le type de la valeur de retour est celui de l'opérande de gauche et que le type de cet opérande n'est pas une classe définie par l'utilisateur (par exemple si c'est un type prédéfini). En effet, on ne peut pas définir l'opérateur à l'intérieur de la classe du premier opérande dans ce cas, puisque cette classe est déjà définie. De même, cette syntaxe peut être utile dans le cas de l'écriture d'opérateurs optimisés pour certains types de données, pour lesquels les opérations réalisées par l'opérateur sont plus simples que celles qui auraient été effectuées après transtypage.

Par exemple, si l'on veut optimiser la multiplication à gauche par un scalaire pour la classe complexe, on devra procéder comme suit :

```
complexe operator*(double k, const complexe &c)
{
    complexe result(c.re()*k,c.im()*k);
    return result;
}
```

ce qui permettra d'écrire des expressions du type :

```
complexe c1, c2;
double r;
...
c1 = r*c2;
```

La première syntaxe n'aurait permis d'écrire un tel opérateur que pour la multiplication à droite par un double. En effet, pour écrire un opérateur interne permettant de réaliser cette optimisation, il aurait fallu surcharger l'opérateur de multiplication de la classe double pour lui faire accepter un objet de type complexe en second opérande...

7.11.3. Opérateurs d'affectation

Nous avons déjà vu un exemple d'opérateur d'affectation avec la classe complexe ci-dessus. Cet opérateur était très simple, mais ce n'est généralement pas toujours le cas, et l'implémentation des opérateurs d'affectation peut parfois soulever quelques problèmes.

Premièrement, comme nous l'avons dit dans la Section 7.8.2, le fait de définir un opérateur d'affectation signale souvent que la classe n'a pas une structure simple et que, par conséquent, le constructeur de copie et le destructeur fournis par défaut par le compilateur ne suffisent pas. Il faut donc veiller à respecter la règle des trois, qui stipule que si l'une de ces méthodes est redéfinie, il faut que les trois le soient. Par exemple, si vous ne redéfinissez pas le constructeur de copie, les écritures telles que :

```
classe object = source;
```

ne fonctionneront pas correctement. En effet, c'est le constructeur de copie qui est appelé ici, et non l'opérateur d'affectation comme on pourrait le penser à première vue. De même, les traitements particuliers effectués lors de la copie ou de l'initialisation d'un objet devront être effectués en ordre inverse dans le destructeur de l'objet. Les traitements de destruction consistent généralement à libérer la mémoire et toutes les ressources allouées dynamiquement.

Lorsque l'on écrit un opérateur d'affectation, on a généralement à reproduire, à peu de choses près, le même code que celui qui se trouve dans le constructeur de copie. Il arrive même parfois que l'on doive libérer les ressources existantes avant de faire l'affectation, et donc le code de l'opérateur d'affectation ressemble souvent à la concaténation du code du destructeur et du code du constructeur de copie. Bien entendu, cette duplication de code est gênante et peu élégante. Une solution simple est d'implémenter une fonction de duplication et une fonction de libération des données. Ces deux fonctions, par exemple `reset` et `clone`, pourront être utilisées dans le destructeur, le constructeur de copie et l'opérateur d'affectation. Le programme devient ainsi beaucoup plus simple. Il ne faut généralement pas utiliser l'opérateur d'affectation dans le constructeur de copie, car cela peut poser des problèmes complexes à résoudre. Par exemple, il faut s'assurer que l'opérateur de copie ne cherche pas à utiliser des données membres non initialisées lors de son appel.

Un autre problème important est celui de l'autoaffectation. Non seulement affecter un objet à lui-même est inutile et consommateur de ressources, mais en plus cela peut être dangereux. En effet, l'affectation risque de détruire les données membres de l'objet avant même qu'elles ne soient copiées, ce qui provoquerait en fin de compte simplement la destruction de l'objet ! Une solution simple consiste ici à ajouter un test sur l'objet source en début d'opérateur, comme dans l'exemple suivant :

```
classe &classe::operator=(const classe &source)
{
    if (&source != this)
```

```
    {
        // Traitement de copie des données :
        ...
    }
    return *this;
}
```

Enfin, la copie des données peut lancer une exception et laisser l'objet sur lequel l'affectation se fait dans un état indéterminé. La solution la plus simple dans ce cas est encore de construire une copie de l'objet source en local, puis d'échanger le contenu des données de l'objet avec cette copie. Ainsi, si la copie échoue pour une raison ou une autre, l'objet source n'est pas modifié et reste dans un état stable. Le pseudo-code permettant de réaliser ceci est le suivant :

```
classe &classe::operator=(const classe &source)
{
    // Construit une copie temporaire de la source :
    class Temp(source);
    // Échange le contenu de cette copie avec l'objet courant :
    swap(Temp, *this);
    // Renvoie l'objet courant (modifié) et détruit les données
    // de la variable temporaire (contenant les anciennes données) :
    return *this;
}
```

Note : Le problème de l'état des objets n'est pas spécifique à l'opérateur d'affectation, mais à toutes les méthodes qui modifient l'objet, donc, en pratique, à toutes les méthodes non `const`. L'écriture de classes sûres au niveau de la gestion des erreurs est donc relativement difficile.

Vous trouverez de plus amples informations sur le mécanisme des exceptions en C++ dans le Chapitre 8.

7.11.4. Opérateurs de transtypage

Nous avons vu dans la Section 7.8.3 que les constructeurs peuvent être utilisés pour convertir des objets du type de leur paramètre vers le type de leur classe. Ces conversions peuvent avoir lieu de manière implicite ou non, selon que le mot clé `explicit` est appliqué au constructeur en question.

Cependant, il n'est pas toujours faisable d'écrire un tel constructeur. Par exemple, la classe cible peut parfaitement être une des classes de la bibliothèque standard, dont on ne doit évidemment pas modifier les fichiers source, ou même un des types de base du langage, pour lequel il n'y a pas de définition. Heureusement, les conversions peuvent malgré tout être réalisées dans ce cas, simplement en surchargeant les opérateurs de transtypage.

Prenons l'exemple de la classe chaine, qui permet de faire des chaînes de caractères dynamiques (de longueur variable). Il est possible de les convertir en chaîne C classiques (c'est-à-dire en tableau de caractères) si l'opérateur `(char const *)` a été surchargé :

```
chaine::operator char const *(void) const;
```

On constatera que cet opérateur n'attend aucun paramètre, puisqu'il s'applique à l'objet qui l'appelle, mais surtout il n'a pas de type. En effet, puisque c'est un opérateur de transtypage, son type est nécessairement celui qui lui correspond (dans le cas présent, char const *).

> **Note :** Si un constructeur de transtypage est également défini dans la classe du type cible de la conversion, il peut exister deux moyens de réaliser le transtypage. Dans ce cas, le compilateur choisira toujours le constructeur de transtypage de la classe cible à la place de l'opérateur de transtypage, sauf s'il est déclaré `explicit`. Ce mot clé peut donc être utilisé partout où l'on veut éviter que le compilateur n'utilise le constructeur de transtypage. Cependant, cette technique ne fonctionne qu'avec les conversions implicites réalisées par le compilateur. Si l'utilisateur effectue un transtypage explicite, ce sera à nouveau le constructeur qui sera appelé.
>
> De plus, les conversions réalisées par l'intermédiaire d'un constructeur sont souvent plus performantes que celles réalisées par l'intermédiaire d'un opérateur de transtypage, en raison du fait que l'on évite ainsi la copie de la variable temporaire dans le retour de l'opérateur de transtypage. On évitera donc de définir les opérateurs de transtypage autant que faire se peut, et on écrira de préférence des constructeurs dans les classes des types cibles des conversions réalisées.

7.11.5. Opérateurs de comparaison

Les opérateurs de comparaison sont très simples à surcharger. La seule chose essentielle à retenir est qu'ils renvoient une valeur booléenne. Ainsi, pour la classe chaine, on peut déclarer les opérateurs d'égalité et d'infériorité (dans l'ordre lexicographique par exemple) de deux chaînes de caractères comme suit :

```
bool chaine::operator==(const chaine &) const;
bool chaine::operator<(const chaine &) const;
```

7.11.6. Opérateurs d'incrémentation et de décrémentation

Les opérateurs d'incrémentation et de décrémentation sont tous les deux doubles, c'est-à-dire que la même notation représente deux opérateurs en réalité. En effet, ils n'ont pas la même signification, selon qu'ils sont placés avant ou après leur opérande. Le problème est que comme ces opérateurs ne prennent pas de paramètres (ils ne travaillent que sur l'objet), il est impossible de les différencier par surcharge. La solution qui a été adoptée est de les différencier en donnant un paramètre fictif de type int à l'un d'entre eux. Ainsi, les opérateurs ++ et −− ne prennent pas de paramètre lorsqu'il s'agit des opérateurs préfixés, et ont un argument fictif (que l'on ne doit pas utiliser) lorsqu'ils sont suffixés. Les versions préfixées des opérateurs doivent renvoyer une référence sur l'objet lui-même, les versions suffixées en revanche peuvent se contenter de renvoyer la valeur de l'objet.

Exemple 7-18. Opérateurs d'incrémentation et de décrémentation

```
class Entier
{
    int i;

public:
    Entier(int j)
```

```
    {
        i=j;
        return;
    }

    Entier operator++(int)   // Opérateur suffixe :
    {                        // retourne la valeur et incrémente
        Entier tmp(i);       // la variable.
        ++i;
        return tmp;
    }

    Entier &operator++(void) // Opérateur préfixe : incrémente
    {                        // la variable et la retourne.
        ++i;
        return *this;
    }
};
```

Note : Les opérateurs suffixés créant des objets temporaires, ils peuvent nuire gravement aux performances des programmes qui les utilisent de manière inconsidérée. Par conséquent, on ne les utilisera que lorsque cela est réellement nécessaire. En particulier, on évitera d'utiliser ces opérateurs dans toutes les opérations d'incrémentation des boucles d'itération.

7.11.7. Opérateur fonctionnel

L'opérateur d'appel de fonctions `()` peut également être surchargé. Cet opérateur permet de réaliser des objets qui se comportent comme des fonctions (ce que l'on appelle des *foncteurs*). La bibliothèque standard C++ en fait un usage intensif, comme nous pourrons le constater dans la deuxième partie de ce document.

L'opérateur fonctionnel est également très utile en raison de son n-arité (`*`, `/`, etc. sont des opérateurs binaires car ils ont deux opérandes, `?:` est un opérateur ternaire car il a trois opérandes, `()` est n-aire car il peut avoir n opérandes). Il est donc utilisé couramment pour les classes de gestion de matrices de nombres, afin d'autoriser l'écriture « `matrice(i,j,k)` ».

Exemple 7-19. Implémentation d'une classe matrice

```
class matrice
{
    typedef double *ligne;
    ligne *lignes;
    unsigned short int n;   // Nombre de lignes (1er paramètre).
    unsigned short int m;   // Nombre de colonnes (2ème paramètre).

public:
    matrice(unsigned short int nl, unsigned short int nc);
    matrice(const matrice &source);
    ~matrice(void);
    matrice &operator=(const matrice &m1);
    double &operator()(unsigned short int i, unsigned short int j);
    double operator()(unsigned short int i, unsigned short int j) const;
};
```

```
// Le constructeur :
matrice::matrice(unsigned short int nl, unsigned short int nc)
{
    n = nl;
    m = nc;
    lignes = new ligne[n];
    for (unsigned short int i=0; i<n; ++i)
        lignes[i] = new double[m];
    return;
}

// Le constructeur de copie :
matrice::matrice(const matrice &source)
{
    m = source.m;
    n = source.n;
    lignes = new ligne[n];   // Alloue.
    for (unsigned short int i=0; i<n; ++i)
    {
        lignes[i] = new double[m];
        for (unsigned short int j=0; j<m; ++j)  // Copie.
            lignes[i][j] = source.lignes[i][j];
    }
    return;
}

// Le destructeur :
matrice::~matrice(void)
{
    for (unsigned short int i=0; i<n; ++i)
        delete[] lignes[i];
    delete[] lignes;
    return;
}

// L'opérateur d'affectation :
matrice &matrice::operator=(const matrice &source)
{
    if (&source != this)
    {
        if (source.n!=n || source.m!=m)   // Vérifie les dimensions.
        {
            for (unsigned short int i=0; i<n; ++i)
                delete[] lignes[i];
            delete[] lignes;              // Détruit...
            m = source.m;
            n = source.n;
            lignes = new ligne[n];        // et réalloue.
            for (i=0; i<n; ++i) lignes[i] = new double[m];
        }
        for (unsigned short int i=0; i<n; ++i) // Copie.
            for (unsigned short int j=0; j<m; ++j)
                lignes[i][j] = source.lignes[i][j];
    }
    return *this;
}
```

```
// Opérateurs d'accès :
double &matrice::operator()(unsigned short int i,
    unsigned short int j)
{
    return lignes[i][j];
}

double matrice::operator()(unsigned short int i,
    unsigned short int j) const
{
    return lignes[i][j];
}
```

Ainsi, on pourra effectuer la déclaration d'une matrice avec :

```
matrice m(2,3);
```

et accéder à ses éléments simplement avec :

```
m(i,j)=6;
```

On remarquera que l'on a défini deux opérateurs fonctionnels dans l'exemple donné ci-dessus. Le premier renvoie une référence et permet de modifier la valeur d'un des éléments de la matrice. Cet opérateur ne peut bien entendu pas s'appliquer à une matrice constante, même simplement pour lire un élément. C'est donc le deuxième opérateur qui sera utilisé pour lire les éléments des matrices constantes, car il renvoie une valeur et non plus une référence. Le choix de l'opérateur à utiliser est déterminé par la présence du mot clé `const`, qui indique que seul cet opérateur peut être utilisé pour une matrice constante.

> **Note :** Les opérations de base sur les matrices (addition, soustraction, inversion, transposition, etc.) n'ont pas été reportées ici par souci de clarté. La manière de définir ces opérateurs a été présentée dans les sections précédentes.

7.11.8. Opérateurs d'indirection et de déréférencement

L'opérateur de déréférencement `*` permet l'écriture de classes dont les objets peuvent être utilisés dans des expressions manipulant des pointeurs. L'opérateur d'indirection `&` quant à lui, permet de renvoyer une adresse autre que celle de l'objet sur lequel il s'applique. Enfin, l'opérateur de déréférencement et de sélection de membres de structures `->` permet de réaliser des classes qui encapsulent d'autres classes.

Si les opérateurs de déréférencement et d'indirection `&` et `*` peuvent renvoyer une valeur de type quelconque, ce n'est pas le cas de l'opérateur de déréférencement et de sélection de membre `->`. Cet opérateur doit nécessairement renvoyer un type pour lequel il doit encore être applicable. Ce type doit donc soit surcharger l'opérateur `->`, soit être un pointeur sur une structure, union ou classe.

Exemple 7-20. Opérateur de déréférencement et d'indirection

```
// Cette classe est encapsulée par une autre classe :
struct Encapsulee
{
    int i;       // Donnée à accéder.
};

Encapsulee o;    // Objet à manipuler.

// Cette classe est la classe encapsulante :
struct Encapsulante
{
    Encapsulee *operator->(void) const
    {
        return &o;
    }

    Encapsulee *operator&(void) const
    {
        return &o;
    }

    Encapsulee &operator*(void) const
    {
        return o;
    }
};

// Exemple d'utilisation :
void f(int i)
{
    Encapsulante e;
    e->i=2;         // Enregistre 2 dans o.i.
    (*e).i = 3;     // Enregistre 3 dans o.i.
    Encapsulee *p = &e;
    p->i = 4;       // Enregistre 4 dans o.i.
    return ;
}
```

7.11.9. Opérateurs d'allocation dynamique de mémoire

Les opérateurs les plus difficiles à écrire sont sans doute les opérateurs d'allocation dynamique de mémoire. Ces opérateurs prennent un nombre variable de paramètres, parce qu'ils sont complètement surchargeables (c'est à dire qu'il est possible de définir plusieurs surcharges de ces opérateurs même au sein d'une même classe, s'ils sont définis de manière interne). Il est donc possible de définir plusieurs opérateurs `new` ou `new[]`, et plusieurs opérateurs `delete` ou `delete[]`. Cependant, les premiers paramètres de ces opérateurs doivent toujours être la taille de la zone de la mémoire à allouer dans le cas des opérateurs `new` et `new[]`, et le pointeur sur la zone de la mémoire à restituer dans le cas des opérateurs `delete` et `delete[]`.

La forme la plus simple de `new` ne prend qu'un paramètre : le nombre d'octets à allouer, qui vaut toujours la taille de l'objet à construire. Il doit renvoyer un pointeur du type void. L'opérateur `delete` correspondant peut prendre, quant à lui, soit un, soit deux paramètres. Comme on l'a déjà dit, le premier paramètre est toujours un pointeur du type void sur l'objet à détruire. Le deuxième paramètre,

s'il existe, est du type size_t et contient la taille de l'objet à détruire. Les mêmes règles s'appliquent pour les opérateurs `new[]` et `delete[]`, utilisés pour les tableaux.

Lorsque les opérateurs `delete` et `delete[]` prennent deux paramètres, le deuxième paramètre est la taille de la zone de la mémoire à restituer. Cela signifie que le compilateur se charge de mémoriser cette information. Pour les opérateurs `new` et `delete`, cela ne cause pas de problème, puisque la taille de cette zone est fixée par le type de l'objet. En revanche, pour les tableaux, la taille du tableau doit être stockée avec le tableau. En général, le compilateur utilise un en-tête devant le tableau d'objets. C'est pour cela que la taille à allouer passée à `new[]`, qui est la même que la taille à désallouer passée en paramètre à `delete[]`, n'est pas égale à la taille d'un objet multipliée par le nombre d'objets du tableau. Le compilateur demande un peu plus de mémoire, pour mémoriser la taille du tableau. On ne peut donc pas, dans ce cas, faire d'hypothèses quant à la structure que le compilateur donnera à la mémoire allouée pour stocker le tableau.

En revanche, si `delete[]` ne prend en paramètre que le pointeur sur le tableau, la mémorisation de la taille du tableau est à la charge du programmeur. Dans ce cas, le compilateur donne à `new[]` la valeur exacte de la taille du tableau, à savoir la taille d'un objet multipliée par le nombre d'objets dans le tableau.

Exemple 7-21. Détermination de la taille de l'en-tête des tableaux

```
#include <stdlib.h>
#include <stdio.h>

int buffer[256];     // Buffer servant à stocker le tableau.

class Temp
{
    char i[13];      // sizeof(Temp) doit être premier.

public:
    static void *operator new[](size_t taille)
    {
       return buffer;
    }

    static void operator delete[](void *p, size_t taille)
    {
       printf("Taille de l'en-tête : %d\n",
           taille-(taille/sizeof(Temp))*sizeof(Temp));
       return ;
    }
};

int main(void)
{
    delete[] new Temp[1];
    return EXIT_SUCCESS;
}
```

Il est à noter qu'aucun des opérateurs `new`, `delete`, `new[]` et `delete[]` ne reçoit le pointeur `this` en paramètre : ce sont des opérateurs statiques. Cela est normal puisque, lorsqu'ils s'exécutent, soit l'objet n'est pas encore créé, soit il est déjà détruit. Le pointeur `this` n'existe donc pas encore (ou n'est plus valide) lors de l'appel de ces opérateurs.

Les opérateurs `new` et `new[]` peuvent avoir une forme encore un peu plus compliquée, qui permet de leur passer des paramètres lors de l'allocation de la mémoire. Les paramètres supplémentaires doivent

impérativement être les paramètres deux et suivants, puisque le premier paramètre indique toujours la taille de la zone de mémoire à allouer.

Comme le premier paramètre est calculé par le compilateur, il n'y a pas de syntaxe permettant de le passer aux opérateurs `new` et `new[]`. En revanche, une syntaxe spéciale est nécessaire pour passer les paramètres supplémentaires. Cette syntaxe est détaillée ci-dessous.

Si l'opérateur `new` est déclaré de la manière suivante dans la classe classe :

```
static void *operator new(size_t taille, paramètres);
```

où `taille` est la taille de la zone de mémoire à allouer et `paramètres` la liste des paramètres additionnels, alors on doit l'appeler avec la syntaxe suivante :

```
new(paramètres) classe;
```

Les paramètres sont donc passés entre parenthèses comme pour une fonction normale. Le nom de la fonction est `new`, et le nom de la classe suit l'expression `new` comme dans la syntaxe sans paramètres. Cette utilisation de `new` est appelée *new avec placement*.

Le placement est souvent utilisé afin de réaliser des réallocations de mémoire d'un objet à un autre. Par exemple, si l'on doit détruire un objet alloué dynamiquement et en reconstruire immédiatement un autre du même type, les opérations suivantes se déroulent :

1. appel du destructeur de l'objet (réalisé par l'expression `delete`) ;
2. appel de l'opérateur `delete` ;
3. appel de l'opérateur `new` ;
4. appel du constructeur du nouvel objet (réalisé par l'expression `new`).

Cela n'est pas très efficace, puisque la mémoire est restituée pour être allouée de nouveau immédiatement après. Il est beaucoup plus logique de réutiliser la mémoire de l'objet à détruire pour le nouvel objet, et de reconstruire ce dernier dans cette mémoire. Cela peut se faire comme suit :

1. appel explicite du destructeur de l'objet à détruire ;
2. appel de `new` avec comme paramètre supplémentaire le pointeur sur l'objet détruit ;
3. appel du constructeur du deuxième objet (réalisé par l'expression `new`).

L'appel de `new` ne fait alors aucune allocation : on gagne ainsi beaucoup de temps.

Exemple 7-22. Opérateurs new avec placement

```
#include <stdlib.h>

class A
{
public:
    A(void)           // Constructeur.
    {
        return ;
    }
```

```
    ~A(void)          // Destructeur.
    {
        return ;
    }

    // L'opérateur new suivant utilise le placement.
    // Il reçoit en paramètre le pointeur sur le bloc
    // à utiliser pour la requête d'allocation dynamique
    // de mémoire.
    static void *operator new (size_t taille, A *bloc)
    {
        return (void *) bloc;
    }

    // Opérateur new normal :
    static void *operator new(size_t taille)
    {
        // Implémentation :
        return malloc(taille);
    }

    // Opérateur delete normal :
    static void operator delete(void *pBlock)
    {
        free(pBlock);
        return ;
    }
};

int main(void)
{
    A *pA=new A;      // Création d'un objet de classe A.
                      // L'opérateur new global du C++ est utilisé.
    pA->~A();         // Appel explicite du destructeur de A.
    A *pB=new(pA) A;  // Réutilisation de la mémoire de A.
    delete pB;        // Destruction de l'objet.
    return EXIT_SUCCESS;
}
```

Dans cet exemple, la gestion de la mémoire est réalisée par les opérateurs `new` et `delete` normaux. Cependant, la réutilisation de la mémoire allouée se fait grâce à un opérateur `new` avec placement, défini pour l'occasion. Ce dernier ne fait strictement rien d'autre que de renvoyer le pointeur qu'on lui a passé en paramètre. On notera qu'il est nécessaire d'appeler explicitement le destructeur de la classe A avant de réutiliser la mémoire de l'objet, car aucune expression delete ne s'en charge avant la réutilisation de la mémoire.

> **Note :** Les opérateurs `new` et `delete` avec placement prédéfinis par la bibliothèque standard C++ effectuent exactement ce que les opérateurs de cet exemple font. Il n'est donc pas nécessaire de les définir, si on ne fait aucun autre traitement que de réutiliser le bloc mémoire que l'opérateur `new` reçoit en paramètre.

Il est impossible de passer des paramètres à l'opérateur `delete` dans une expression `delete`. Cela est dû au fait qu'en général on ne connaît pas le contexte de la destruction d'un objet (alors qu'à l'allocation, on connaît le contexte de création de l'objet). Normalement, il ne peut donc y avoir

qu'un seul opérateur `delete`. Cependant, il existe un cas où l'on connaît le contexte de l'appel de l'opérateur `delete` : c'est le cas où le constructeur de la classe lance une exception (voir le Chapitre 8 pour plus de détails à ce sujet). Dans ce cas, la mémoire allouée par l'opérateur `new` doit être restituée et l'opérateur `delete` est automatiquement appelé, puisque l'objet n'a pas pu être construit. Afin d'obtenir un comportement symétrique, il est permis de donner des paramètres additionnels à l'opérateur `delete`. Lorsqu'une exception est lancée dans le constructeur de l'objet alloué, l'opérateur `delete` appelé est l'opérateur dont la liste des paramètres correspond à celle de l'opérateur `new` qui a été utilisé pour créer l'objet. Les paramètres passés à l'opérateur `delete` prennent alors exactement les mêmes valeurs que celles qui ont été données aux paramètres de l'opérateur `new` lors de l'allocation de la mémoire de l'objet. Ainsi, si l'opérateur `new` a été utilisé sans placement, l'opérateur `delete` sans placement sera appelé. En revanche, si l'opérateur `new` a été appelé avec des paramètres, l'opérateur `delete` qui a les mêmes paramètres sera appelé. Si aucun opérateur `delete` ne correspond, aucun opérateur `delete` n'est appelé (si l'opérateur `new` n'a pas alloué de mémoire, cela n'est pas grave, en revanche, si de la mémoire a été allouée, elle ne sera pas restituée). Il est donc important de définir un opérateur `delete` avec placement pour chaque opérateur `new` avec placement défini. L'exemple précédent doit donc être réécrit de la manière suivante :

```
#include <stdlib.h>

static bool bThrow = false;

class A
{
public:
    A(void)           // Constructeur.
    {
        // Le constructeur est susceptible
        // de lancer une exception :
 if (bThrow) throw 2;
        return ;
    }

    ~A(void)          // Destructeur.
    {
        return ;
    }

    // L'opérateur new suivant utilise le placement.
    // Il reçoit en paramètre le pointeur sur le bloc
    // à utiliser pour la requête d'allocation dynamique
    // de mémoire.
    static void *operator new (size_t taille, A *bloc)
    {
        return (void *) bloc;
    }

    // L'opérateur delete suivant est utilisé dans les expressions
    // qui utilisent l'opérateur new avec placement ci-dessus,
    // si une exception se produit dans le constructeur.
    static void operator delete(void *p, A *bloc)
    {
        // On ne fait rien, parce que l'opérateur new correspondant
        // n'a pas alloué de mémoire.
        return ;
    }
```

```
    // Opérateur new et delete normaux :
    static void *operator new(size_t taille)
    {
        return malloc(taille);
    }

    static void operator delete(void *pBlock)
    {
        free(pBlock);
        return ;
    }
};

int main(void)
{
    A *pA=new A;      // Création d'un objet de classe A.
    pA->~A();         // Appel explicite du destructeur de A.
    bThrow = true;    // Maintenant, le constructeur de A lance
                      // une exception.
    try
    {
        A *pB=new(pA) A;  // Réutilisation de la mémoire de A.
                          // Si une exception a lieu, l'opérateur
                          // delete(void *, A *) avec placement
                          // est utilisé.
        delete pB;        // Destruction de l'objet.
    }
    catch (...)
    {
        // L'opérateur delete(void *, A *) ne libère pas la mémoire
        // allouée lors du premier new. Il faut donc quand même
        // le faire, mais sans delete, car l'objet pointé par pA
        // est déjà détruit, et celui pointé par pB l'a été par
        // l'opérateur delete(void *, A *) :
        free(pA);
    }
    return EXIT_SUCCESS;
}
```

Note : Il est possible d'utiliser le placement avec les opérateurs `new[]` et `delete[]` exactement de la même manière qu'avec les opérateurs `new` et `delete`.

On notera que lorsque l'opérateur `new` est utilisé avec placement, si le deuxième argument est de type size_t, l'opérateur `delete` à deux arguments peut être interprété soit comme un opérateur `delete` classique sans placement mais avec deux paramètres, soit comme l'opérateur `delete` avec placement correspondant à l'opérateur `new` avec placement. Afin de résoudre cette ambiguïté, le compilateur interprète systématiquement l'opérateur `delete` avec un deuxième paramètre de type size_t comme étant l'opérateur à deux paramètres sans placement. Il est donc impossible de définir un opérateur `delete` avec placement s'il a deux paramètres, le deuxième étant de type size_t. Il en est de même avec les opérateurs `new[]` et `delete[]`.

Quelle que soit la syntaxe que vous désirez utiliser, les opérateurs `new`, `new[]`, `delete` et `delete[]` doivent avoir un comportement bien déterminé. En particulier, les opérateurs `delete` et `delete[]`

doivent pouvoir accepter un pointeur nul en paramètre. Lorsqu'un tel pointeur est utilisé dans une expression `delete`, aucun traitement ne doit être fait.

Enfin, vos opérateurs `new` et `new[]` doivent, en cas de manque de mémoire, appeler un gestionnaire d'erreur. Le gestionnaire d'erreur fourni par défaut lance une exception de classe std::bad_alloc (voir le Chapitre 8 pour plus de détails sur les exceptions). Cette classe est définie comme suit dans le fichier d'en-tête `new` :

```
class bad_alloc : public exception
{
public:
    bad_alloc(void) throw();
    bad_alloc(const bad_alloc &) throw();
    bad_alloc &operator=(const bad_alloc &) throw();
    virtual ~bad_alloc(void) throw();
    virtual const char *what(void) const throw();
};
```

> **Note :** Comme son nom l'indique, cette classe est définie dans l'espace de nommage `std::`. Si vous ne voulez pas utiliser les notions des espaces de nommage, vous devrez inclure le fichier d'en-tête `new.h` au lieu de `new`. Vous obtiendrez de plus amples renseignements sur les espaces de nommage dans le Chapitre 10.

La classe exception dont bad_alloc hérite est déclarée comme suit dans le fichier d'en-tête `exception` :

```
class exception
{
public:
    exception (void) throw();
    exception(const exception &) throw();
    exception &operator=(const exception &) throw();
    virtual ~exception(void) throw();
    virtual const char *what(void) const throw();
};
```

> **Note :** Contrairement aux en-têtes de la bibliothèque C standard, les noms des en-têtes de la bibliothèque standard C++ n'ont pas d'extension. Cela permet notamment d'éviter des conflits de noms avec les en-têtes existants.
>
> Vous trouverez plus d'informations sur les exceptions dans le Chapitre 8.

Si vous désirez remplacer le gestionnaire par défaut, vous pouvez utiliser la fonction `std::set_new_handler`. Cette fonction attend en paramètre le pointeur sur le gestionnaire d'erreur à installer et renvoie le pointeur sur le gestionnaire d'erreur précédemment installé. Les gestionnaires d'erreurs ne prennent aucun paramètre et ne renvoient aucune valeur. Leur comportement doit être le suivant :

- soit ils prennent les mesures nécessaires pour permettre l'allocation du bloc de mémoire demandé et rendent la main à l'opérateur `new`. Ce dernier refait alors une tentative pour allouer le bloc de mémoire. Si cette tentative échoue à nouveau, le gestionnaire d'erreur est rappelé. Cette boucle se poursuit jusqu'à ce que l'opération se déroule correctement ou qu'une exception std::bad_alloc soit lancée ;
- soit ils lancent une exception de classe std::bad_alloc ;
- soit ils terminent l'exécution du programme en cours.

La bibliothèque standard définit une version avec placement des opérateurs `new` et `new[]`, qui renvoient le pointeur nul au lieu de lancer une exception en cas de manque de mémoire. Ces opérateurs prennent un deuxième paramètre, de type std::nothrow_t, qui doit être spécifié lors de l'appel. La bibliothèque standard définit un objet constant de ce type afin que les programmes puissent l'utiliser sans avoir à le définir eux-même. Cet objet se nomme `std::nothrow`

Exemple 7-23. Utilisation de new sans exception

```
char *data = new(std::nothrow) char[25];
if (data == NULL)
{
    // Traitement de l'erreur...
    ⋮
}
```

Note : La plupart des compilateurs ne respectent pas les règles dictées par la norme C++. En effet, ils préfèrent retourner la valeur nulle en cas de manque de mémoire au lieu de lancer une exception. On peut rendre ces implémentations compatibles avec la norme en installant un gestionnaire d'erreur qui lance lui-même l'exception std::bad_alloc.

7.12. Des entrées - sorties simplifiées

Les flux d'entrée / sortie de la bibliothèque standard C++ constituent sans doute l'une des applications les plus intéressantes de la surcharge des opérateurs. Comme nous allons le voir, la surcharge des opérateurs << et >> permet d'écrire et de lire sur ces flux de manière très intuitive.

En effet, la bibliothèque standard C++ définit dans l'en-tête `iostream` des classes extrêmement puissantes permettant de manipuler les flux d'entrée / sortie. Ces classes réalisent en particulier les opérations d'entrée / sortie de et vers les périphériques d'entrée et les périphériques de sortie standards (généralement, le clavier et l'écran), mais elles ne s'arrêtent pas là : elles permettent également de travailler sur des fichiers ou encore sur des tampons en mémoire.

Les classes d'entrée / sortie de la bibliothèque standard C++ permettent donc d'effectuer les mêmes opérations que les fonctions `printf` et `scanf` de la bibliothèque C standard. Cependant, grâce au mécanisme de surcharge des opérateurs, elles sont beaucoup plus faciles d'utilisation. En effet, les opérateurs << et >> de ces classes ont été surchargés pour chaque type de donnée du langage, permettant ainsi de réaliser des entrées / sorties typées extrêmement facilement. L'opérateur <<, également appelée opérateur d'insertion, sera utilisé pour réaliser des écritures sur un flux de données, tandis que l'opérateur >>, ou opérateur d'extraction, permettra de réaliser la lecture d'une nouvelle donnée dans

le flux d'entrée. Ces deux opérateurs renvoient tous les deux le flux de données utilisé, ce qui permet de réaliser plusieurs opérations d'entrée / sortie successivement sur le même flux.

> **Note :** Cette section n'a pas pour but de décrire en détail les flux d'entrée / sortie de la bibliothèque standard C++, mais plutôt d'en faire une présentation simple permettant de les utiliser sans avoir à se plonger prématurément dans des notions extrêmement évoluées. Vous trouverez une description exhaustive des mécanismes des flux d'entrée / sortie de la bibliothèque standard C++ dans le Chapitre 15.

La bibliothèque standard définit quatre instances particulières de ses classes d'entrée / sortie : `cin`, `cout`, `cerr` et `clog`. Ces objets sont des instances des classes istream et ostream, prenant respectivement en charge l'entrée et la sortie des données des programmes. L'objet `cin` correspond au flux d'entrée standard `stdin` du programme, et l'objet `cout` aux flux de sortie standard `stdout`. Enfin, les objets `cerr` et `clog` sont associés au flux d'erreurs standard `stderr`. Théoriquement, `cerr` doit être utilisé pour l'écriture des messages d'erreur des programmes, et `clog` pour les messages d'information. Cependant, en pratique, les données écrites sur ces deux flux sont écrites dans le même flux, et l'emploi de l'objet `clog` est assez rare.

L'utilisation des opérateurs d'insertion et d'extraction sur ces flux se résume donc à la syntaxe suivante :

```
cin >> variable [>> variable [...]];
cout << valeur [<< valeur [...]];
```

Comme on le voit, il est possible d'effectuer plusieurs entrées ou plusieurs sortie successivement sur un même flux.

De plus, la bibliothèque standard définie ce que l'on appelle des *manipulateurs* permettant de réaliser des opérations simples sur les flux d'entrée / sortie. Le manipulateur le plus utilisé est sans nul doute le manipulateur `endl` qui, comme son nom l'indique, permet de signaler une fin de ligne et d'effectuer un saut de ligne lorsqu'il est employé sur un flux de sortie.

Exemple 7-24. Flux d'entrée / sortie cin et cout

```
#include <stdlib.h>
#include <iostream>
using namespace std;

int main(void)
{
    int i;
    // Lit un entier :
    cin >> i;
    // Affiche cet entier et le suivant :
    cout << i << " " << i+1 << endl;
    return EXIT_SUCCESS;
}
```

> **Note :** Comme on le verra dans le Chapitre 15, les manipulateurs sont en réalité des fonctions pour le type desquelles un opérateur `<<` ou un opérateur `>>` a été défini dans les classes d'entrée / sortie. Ces opérateurs appellent ces fonctions, qui effectuent chacune des modifications spécifiques sur le flux sur lequel elles travaillent.
>
> Les flux d'entrée / sortie `cin`, `cout` `cerr` et `clog` sont déclarés dans l'espace de nommage `std::` de la bibliothèque standard C++. On devra donc faire précéder leur nom du préfixe `std::` pour

y accéder, ou utiliser un directive `using` pour importer les symboles de la bibliothèque standard C++ dans l'espace de nommage global. Vous trouverez de plus amples renseignements sur les espaces de nommages dans le Chapitre 10.

Les avantages des flux C++ sont nombreux, on notera en particulier ceux-ci :

- le type des donnée est automatiquement pris en compte par les opérateurs d'insertion et d'extraction (ils sont surchargés pour tous les types prédéfinis) ;
- les opérateurs d'extraction travaillent par référence (on ne risque plus d'omettre l'opérateur & dans la fonction `scanf`) ;
- il est possible de définir des opérateurs d'insertion et d'extraction pour d'autres types de données que les types de base du langage ;
- leur utilisation est globalement plus simple.

Les flux d'entrée / sortie définis par la bibliothèque C++ sont donc d'une extrême souplesse et sont extensibles aux types de données utilisateur. Par ailleurs, ils disposent d'un grand nombre de paramètres de formatage et d'options avancées. Toutes ces fonctionnalités seront décrites dans le Chapitre 15, où nous verrons également comment réaliser des entrées / sorties dans des fichiers.

7.13. Méthodes virtuelles

Les *méthodes virtuelles* n'ont strictement rien à voir avec les classes virtuelles, bien qu'elles utilisent le même mot clé `virtual`. Ce mot clé est utilisé ici dans un contexte et dans un sens différent.

Nous savons qu'il est possible de redéfinir les méthodes d'une classe mère dans une classe fille. Lors de l'appel d'une fonction ainsi redéfinie, la fonction appelée est la dernière fonction définie dans la hiérarchie de classe. Pour appeler la fonction de la classe mère alors qu'elle a été redéfinie, il faut préciser le nom de la classe à laquelle elle appartient avec l'opérateur de résolution de portée (`::`).

Bien que simple, cette utilisation de la redéfinition des méthodes peut poser des problèmes. Supposons qu'une classe B hérite de sa classe mère A. Si A possède une méthode `x` appelant une autre méthode `y` redéfinie dans la classe fille B, que se passe-t-il lorsqu'un objet de classe B appelle la méthode `x` ? La méthode appelée étant celle de la classe A, elle appellera la méthode `y` de la classe A. Par conséquent, la redéfinition de `y` ne sert à rien dès qu'on l'appelle à partir d'une des fonctions d'une des classes mères.

Une première solution consisterait à redéfinir la méthode `x` dans la classe B. Mais ce n'est ni élégant, ni efficace. Il faut en fait forcer le compilateur à ne pas faire le lien dans la fonction `x` de la classe A avec la fonction `y` de la classe A. Il faut que `x` appelle soit la fonction `y` de la classe A si elle est appelée par un objet de la classe A, soit la fonction `y` de la classe B si elle est appelée pour un objet de la classe B. Le lien avec l'une des méthodes `y` ne doit être fait qu'au moment de l'exécution, c'est-à-dire qu'on doit faire une édition de liens dynamique.

Le C++ permet de faire cela. Pour cela, il suffit de déclarer virtuelle la fonction de la classe de base qui est redéfinie dans la classe fille, c'est-à-dire la fonction `y`. Cela se fait en faisant précéder par le mot clé `virtual` dans la classe de base.

Exemple 7-25. Redéfinition de méthode de classe de base

```
#include <iostream>

using namespace std;

// Définit la classe de base des données.

class DonneeBase
{
protected:
    int Numero;   // Les données sont numérotées.
    int Valeur;   // et sont constituées d'une valeur entière
                  // pour les données de base.
public:
    void Entre(void);       // Entre une donnée.
    void MiseAJour(void);   // Met à jour la donnée.
};

void DonneeBase::Entre(void)
{
    cin >> Numero;          // Entre le numéro de la donnée.
    cout << endl;
    cin >> Valeur;          // Entre sa valeur.
    cout << endl;
    return;
}

void DonneeBase::MiseAJour(void)
{
    Entre();                // Entre une nouvelle donnée
                            // à la place de la donnée en cours.
    return;
}

/* Définit la classe des données détaillées. */

class DonneeDetaillee : private DonneeBase
{
    int ValeurEtendue;      // Les données détaillées ont en plus
                            // une valeur étendue.

public:
    void Entre(void);       // Redéfinition de la méthode d'entrée.
};

void DonneeDetaillee::Entre(void)
{
    DonneeBase::Entre();    // Appelle la méthode de base.
    cin >> ValeurEtendue;  // Entre la valeur étendue.
    cout << endl;
    return;
}
```

Si `d` est un objet de la classe DonneeDetaillee, l'appel de `d.Entre` ne causera pas de problème. En revanche, l'appel de `d.MiseAJour` ne fonctionnera pas correctement, car la fonction `Entre` appelée

dans `MiseAJour` est la fonction de la classe DonneeBase, et non la fonction redéfinie dans DonneeDetaille.

Il fallait déclarer la fonction `Entre` comme une fonction virtuelle. Il n'est nécessaire de le faire que dans la classe de base. Celle-ci doit donc être déclarée comme suit :

```
class DonneeBase
{
protected:
    int Numero;
    int Valeur;

public:
    virtual void Entre(void);   // Fonction virtuelle.
    void MiseAJour(void);
};
```

Cette fois, la fonction `Entre` appelée dans `MiseAJour` est soit la fonction de la classe DonneeBase, si `MiseAJour` est appelée pour un objet de classe DonneeBase, soit celle de la classe DonneeDetaille si `MiseAJour` est appelée pour un objet de la classe DonneeDetaillee.

En résumé, les méthodes virtuelles sont des méthodes qui sont appelées selon la vraie classe de l'objet qui l'appelle. Les objets qui contiennent des méthodes virtuelles peuvent être manipulés en tant qu'objets des classes de base, tout en effectuant les bonnes opérations en fonction de leur type. Ils apparaissent donc comme étant des objets de la classe de base et des objets de leur classe complète indifféremment, et on peut les considérer soit comme les uns, soit comme les autres. Un tel comportement est appelé *polymorphisme* (c'est-à-dire qui peut avoir plusieurs aspects différents). Nous verrons une application du polymorphisme dans le cas des pointeurs sur les objets.

7.14. Dérivation

Nous allons voir ici les *règles de dérivation*. Ces règles permettent de savoir ce qui est autorisé et ce qui ne l'est pas lorsqu'on travaille avec des classes de base et leurs classes filles (ou classes dérivées).

La première règle, qui est aussi la plus simple, indique qu'il est possible d'utiliser un objet d'une classe dérivée partout où l'on peut utiliser un objet d'une de ses classes mères. Les méthodes et données des classes mères appartiennent en effet par héritage aux classes filles. Bien entendu, on doit avoir les droits d'accès sur les membres de la classe de base que l'on utilise (l'accès peut être restreint lors de l'héritage).

La deuxième règle indique qu'il est possible de faire une affectation d'une classe dérivée vers une classe mère. Les données qui ne servent pas à l'initialisation sont perdues, puisque la classe mère ne possède pas les champs correspondants. En revanche, l'inverse est strictement interdit. En effet, les données de la classe fille qui n'existent pas dans la classe mère ne pourraient pas recevoir de valeur, et l'initialisation ne se ferait pas correctement.

Enfin, la troisième règle dit que les pointeurs des classes dérivées sont compatibles avec les pointeurs des classes mères. Cela signifie qu'il est possible d'affecter un pointeur de classe dérivée à un pointeur d'une de ses classes de base. Il faut bien entendu que l'on ait en outre le droit d'accéder à la classe de base, c'est-à-dire qu'au moins un de ses membres puisse être utilisé. Cette condition n'est pas toujours vérifiée, en particulier pour les classes de base dont l'héritage est `private`.

Un objet dérivé pointé par un pointeur d'une des classes mères de sa classe est considéré comme un objet de la classe du pointeur qui le pointe. Les données spécifiques à sa classe ne sont pas supprimées, elles sont seulement momentanément inaccessibles. Cependant, le mécanisme des méthodes virtuelles continue de fonctionner correctement. En particulier, le destructeur de la classe de base doit être déclaré en tant que méthode virtuelle. Cela permet d'appeler le bon destructeur en cas de destruction de l'objet.

Il est possible de convertir un pointeur de classe de base en un pointeur de classe dérivée si la classe de base n'est pas virtuelle. Cependant, même lorsque la classe de base n'est pas virtuelle, cela est dangereux, car la classe dérivée peut avoir des membres qui ne sont pas présents dans la classe de base, et l'utilisation de ce pointeur peut conduire à des erreurs très graves. C'est pour cette raison qu'un transtypage est nécessaire pour ce type de conversion.

Soient par exemple les deux classes définies comme suit :

```
#include <iostream>

using namespace std;

class Mere
{
public:
    Mere(void);
    ~Mere(void);
};

Mere::Mere(void)
{
    cout << "Constructeur de la classe mère." << endl;
    return;
}

Mere::~Mere(void)
{
    cout << "Destructeur de la classe mère." << endl;
    return;
}

class Fille : public Mere
{
public:
    Fille(void);
    ~Fille(void);
};

Fille::Fille(void) : Mere()
{
    cout << "Constructeur de la classe fille." << endl;
    return;
}

Fille::~Fille(void)
{
    cout << "Destructeur de la classe fille." << endl;
    return;
}
```

Avec ces définitions, seule la première des deux affectations suivantes est autorisée :

```
Mere m;   // Instanciation de deux objets.
Fille f;

m=f;      // Cela est autorisé, mais l'inverse ne le serait pas :
f=m;      // ERREUR !! (ne compile pas).
```

Les mêmes règles sont applicables pour les pointeurs d'objets :

```
Mere *pm, m;
Fille *pf, f;
pf=&f;    // Autorisé.
pm=pf;    // Autorisé. Les données et les méthodes
          // de la classe fille ne sont plus accessibles
          // avec ce pointeur : *pm est un objet
          // de la classe mère.
pf=&m;    // ILLÉGAL : il faut faire un transtypage :
pf=(Fille *) &m;  // Cette fois, c'est légal, mais DANGEREUX !
          // En effet, les méthodes de la classe filles
          // ne sont pas définies, puisque m est une classe mère.
```

L'utilisation d'un pointeur sur la classe de base pour accéder à une classe dérivée nécessite d'utiliser des méthodes virtuelles. En particulier, il est nécessaire de rendre virtuels les destructeurs. Par exemple, avec la définition donnée ci-dessus pour les deux classes, le code suivant est faux :

```
Mere *pm;
Fille *pf = new Fille;
pm = pf;
delete pm; // Appel du destructeur de la classe mère !
```

Pour résoudre le problème, il faut que le destructeur de la classe mère soit virtuel (il est inutile de déclarer virtuel le destructeur des classes filles) :

```
class Mere
{
public:
    Mere(void);
    virtual ~Mere(void);
};
```

On notera que bien que l'opérateur `delete` soit une fonction statique, le bon destructeur est appelé, car le destructeur est déclaré `virtual`. En effet, l'opérateur `delete` recherche le destructeur à appeler dans la classe de l'objet le plus dérivé. De plus, l'opérateur `delete` restitue la mémoire de l'objet complet, et pas seulement celle du sous-objet référencé par le pointeur utilisé dans l'expression `delete`. Lorsqu'on utilise la dérivation, il est donc très important de déclarer les destructeurs virtuels pour que l'opérateur `delete` utilise le vrai type de l'objet à détruire.

7.15. Méthodes virtuelles pures - Classes abstraites

Une *méthode virtuelle pure* est une méthode qui est déclarée mais non définie dans une classe. Elle est définie dans une des classes dérivées de cette classe.

Une *classe abstraite* est une classe comportant au moins une méthode virtuelle pure.

Étant donné que les classes abstraites ont des méthodes non définies, il est impossible d'instancier des objets pour ces classes. En revanche, on pourra les référencer avec des pointeurs.

Le mécanisme des méthodes virtuelles pures et des classes abstraites permet de créer des classes de base contenant toutes les caractéristiques d'un ensemble de classes dérivées, pour pouvoir les manipuler avec un unique type de pointeur. En effet, les pointeurs des classes dérivées sont compatibles avec les pointeurs des classes de base, on pourra donc référencer les classes dérivées avec des pointeurs sur les classes de base, donc avec un unique type sous-jacent : celui de la classe de base. Cependant, les méthodes des classes dérivées doivent exister dans la classe de base pour pouvoir être accessibles à travers le pointeur sur la classe de base. C'est ici que les méthodes virtuelles pures apparaissent. Elles forment un moule pour les méthodes des classes dérivées, qui les définissent. Bien entendu, il faut que ces méthodes soient déclarées virtuelles, puisque l'accès se fait avec un pointeur de classe de base et qu'il faut que ce soit la méthode de la classe réelle de l'objet (c'est-à-dire la classe dérivée) qui soit appelée.

Pour déclarer une méthode virtuelle pure dans une classe, il suffit de faire suivre sa déclaration de « `=0` ». La fonction doit également être déclarée virtuelle :

```
virtual type nom(paramètres) =0;
```

`=0` signifie ici simplement qu'il n'y a pas d'implémentation de cette méthode dans cette classe.

> **Note :** `=0` doit être placé complètement en fin de déclaration, c'est-à-dire après le mot clé `const` pour les méthodes `const` et après la déclaration de la liste des exceptions autorisées (voir le Chapitre 8 pour plus de détails à ce sujet).

Un exemple vaut mieux qu'un long discours. Soit donc, par exemple, à construire une structure de données pouvant contenir d'autres structures de données, quels que soient leurs types. Cette structure de données est appelée un conteneur, parce qu'elle contient d'autres structures de données. Il est possible de définir différents types de conteneurs. Dans cet exemple, on ne s'intéressera qu'au conteneur de type sac.

Un *sac* est un conteneur pouvant contenir zéro ou plusieurs objets, chaque objet n'étant pas forcément unique. Un objet peut donc être placé plusieurs fois dans le sac. Un sac dispose de deux fonctions permettant d'y mettre et d'en retirer un objet. Il a aussi une fonction permettant de dire si un objet se trouve dans le sac.

Nous allons déclarer une classe abstraite qui servira de classe de base pour tous les objets utilisables. Le sac ne manipulera que des pointeurs sur la classe abstraite, ce qui permettra son utilisation pour toute classe dérivant de cette classe. Afin de différencier deux objets égaux, un numéro unique devra être attribué à chaque objet manipulé. Le choix de ce numéro est à la charge des objets, la classe abstraite dont ils dérivent devra donc avoir une méthode renvoyant ce numéro. Les objets devront tous pouvoir être affichés dans un format qui leur est propre. La fonction à utiliser pour cela sera `print`. Cette fonction sera une méthode virtuelle pure de la classe abstraite, puisqu'elle devra être définie pour chaque objet.

Passons maintenant au programme...

Exemple 7-26. Conteneur d'objets polymorphiques

```
#include <iostream>

using namespace std;

/*************  LA CLASSE ABSTRAITE DE BASE   *****************/

class Object
{
    unsigned long int new_handle(void);

protected:
    unsigned long int h;         // Identifiant de l'objet.

public:
    Object(void);                // Le constructeur.
    virtual ~Object(void);       // Le destructeur virtuel.
    virtual void print(void) =0; // Fonction virtuelle pure.
    unsigned long int handle(void) const;  // Fonction renvoyant
                                 // le numéro d'identification
                                 // de l'objet.
};

// Cette fonction n'est appelable que par la classe Object :

unsigned long int Object::new_handle(void)
{
    static unsigned long int hc = 0;
    return hc = hc + 1;          // hc est l'identifiant courant.
                                 // Il est incrémenté
}                                // à chaque appel de new_handle.

// Le constructeur de Object doit être appelé par les classes dérivées :

Object::Object(void)
{
    h = new_handle();            // Trouve un nouvel identifiant.
    return;
}

Object::~Object(void)
{
    return ;
}

unsigned long int Object::handle(void) const
{
    return h;                    // Renvoie le numéro de l'objet.
}

/******************** LA CLASSE SAC   *****************/

class Bag : public Object       // La classe sac. Elle hérite
                                // de Object, car un sac peut
                                // en contenir un autre. Le sac
                                // est implémenté sous la forme
```

```
                                // d'une liste chaînée.
{
    struct BagList
    {
        BagList *next;
        Object  *ptr;
    };

    BagList *head;              // La tête de liste.

public:
    Bag(void);        // Le constructeur : appel celui de Object.
    ~Bag(void);       // Le destructeur.
    void print(void); // Fonction d'affichage du sac.
    bool has(unsigned long int) const;
                      // true si le sac contient l'objet.
    bool is_empty(void) const;   // true si le sac est vide.
    void add(Object &);          // Ajoute un objet.
    void remove(Object &);       // Retire un objet.
};

Bag::Bag(void) : Object()
{
    return;  // Ne fait rien d'autre qu'appeler Object::Object().
}

Bag::~Bag(void)
{
    BagList *tmp = head;   // Détruit la liste d'objet.
    while (tmp != NULL)
    {
        tmp = tmp->next;
        delete head;
        head = tmp;
    }
    return;
}

void Bag::print(void)
{
    BagList *tmp = head;
    cout << "Sac n° " << handle() << "." << endl;
    cout << "    Contenu :" << endl;

    while (tmp != NULL)
    {
        cout << "\t";        // Indente la sortie des objets.
        tmp->ptr->print();   // Affiche la liste objets.
        tmp = tmp->next;
    }
    return;
}

bool Bag::has(unsigned long int h) const
{
    BagList *tmp = head;
    while (tmp != NULL && tmp->ptr->handle() != h)
```

```
        tmp = tmp->next;     // Cherche l'objet.
    return (tmp != NULL);
}

bool Bag::is_empty(void) const
{
    return (head==NULL);
}

void Bag::add(Object &o)
{
    BagList *tmp = new BagList;   // Ajoute un objet à la liste.
    tmp->ptr = &o;
    tmp->next = head;
    head = tmp;
    return;
}

void Bag::remove(Object &o)
{
    BagList *tmp1 = head, *tmp2 = NULL;
    while (tmp1 != NULL && tmp1->ptr->handle() != o.handle())
    {
        tmp2 = tmp1;        // Cherche l'objet...
        tmp1 = tmp1->next;
    }
    if (tmp1!=NULL)         // et le supprime de la liste.
    {
        if (tmp2!=NULL) tmp2->next = tmp1->next;
        else head = tmp1->next;
        delete tmp1;
    }
    return;
}
```

Avec la classe Bag définie telle quelle, il est à présent possible de stocker des objets dérivant de la classe Object avec les fonctions `add` et `remove` :

```
#include <stdlib.h>

class MonObjet : public Object
{
    /*  Définir la méthode print() pour l'objet...  */
};

Bag MonSac;

int main(void)
{
    MonObjet a, b, c;    // Effectue quelques opérations
                         // avec le sac :
    MonSac.add(a);
    MonSac.add(b);
    MonSac.add(c);
    MonSac.print();
    MonSac.remove(b);
```

```
    MonSac.add(MonSac);  // Un sac peut contenir un sac !
    MonSac.print();      // Attention ! Cet appel est récursif !
                         // (plantage assuré).
    return EXIT_SUCCESS;
}
```

Nous avons vu que la classe de base servait de moule aux classes dérivées. Le droit d'empêcher une fonction membre virtuelle pure définie dans une classe dérivée d'accéder en écriture non seulement aux données de la classe de base, mais aussi aux données de la classe dérivée, peut donc faire partie de ses prérogatives. Cela est faisable en déclarant le pointeur `this` comme étant un pointeur constant sur objet constant. Nous avons vu que cela pouvait se faire en rajoutant le mot clé `const` après la déclaration de la fonction membre. Par exemple, comme l'identifiant de l'objet de base est placé en `protected` au lieu d'être en `private`, la classe Object autorise ses classes dérivées à le modifier. Cependant, elle peut empêcher la fonction `print` de le modifier en la déclarant `const` :

```
class Object
{
    unsigned long int new_handle(void);

protected:
    unsigned long int h;

public:
    Object(void);                        // Le constructeur.
    virtual void print(void) const=0;  // Fonction virtuelle pure.
    unsigned long int handle(void) const; // Fonction renvoyant
                                         // le numéro d'identification
                                         // de l'objet.
};
```

Dans l'exemple donné ci-dessus, la fonction `print` peut accéder en lecture à *h*, mais plus en écriture. En revanche, les autres fonctions membres des classes dérivées peuvent y avoir accès, puisque c'est une donnée membre `protected`. Cette méthode d'encapsulation est donc coopérative (elle requiert la bonne volonté des autres fonctions membres des classes dérivées), tout comme la méthode qui consistait en C à déclarer une variable constante. Cependant, elle permettra de détecter des anomalies à la compilation, car si une fonction `print` cherche à modifier l'objet sur lequel elle travaille, il y a manifestement une erreur de conception.

Bien entendu, cela fonctionne également avec les fonctions membres virtuelles non pures, et même avec les fonctions non virtuelles.

7.16. Pointeurs sur les membres d'une classe

Nous avons déjà vu les pointeurs sur les objets. Il nous reste à voir les pointeurs sur les membres des classes.

Les classes regroupent les caractéristiques des données et des fonctions des objets. Les membres des classes ne peuvent donc pas être manipulés sans passer par la classe à laquelle ils appartiennent. Par conséquent, il faut, lorsqu'on veut faire un pointeur sur un membre, indiquer le nom de sa classe. Pour cela, la syntaxe suivante est utilisée :

```
définition classe::* pointeur
```

Par exemple, si une classe test contient des entiers, le type de pointeurs à utiliser pour stocker leur adresse est :

```
int test::*
```

Si on veut déclarer un pointeur `p` de ce type, on écrira donc :

```
int test::*p1;   // Construit le pointeur sur entier
                 // de la classe test.
```

Une fois le pointeur déclaré, on pourra l'initialiser en prenant l'adresse du membre de la classe du type correspondant. Pour cela, il faudra encore spécifier le nom de la classe avec l'opérateur de résolution de portée :

```
p1 = &test::i;   // Récupère l'adresse de i.
```

La même syntaxe est utilisable pour les fonctions. L'emploi d'un `typedef` est dans ce cas fortement recommandé. Par exemple, si la classe test dispose d'une fonction membre appelée `lit`, qui n'attend aucun paramètre et qui renvoie un entier, on pourra récupérer son adresse ainsi :

```
typedef int (test::* pf)(void);  // Définit le type de pointeur.
pf p2=&test::lit;                // Construit le pointeur et
                                 // lit l'adresse de la fonction.
```

Cependant, ces pointeurs ne sont pas utilisables directement. En effet, les données d'une classe sont instanciées pour chaque objet, et les fonctions membres reçoivent systématiquement le pointeur `this` sur l'objet de manière implicite. On ne peut donc pas faire un déréférencement direct de ces pointeurs. Il faut spécifier l'objet pour lequel le pointeur va être utilisé. Cela se fait avec la syntaxe suivante :

```
objet.*pointeur
```

Pour les pointeurs d'objet, on pourra utiliser l'opérateur `->*` à la place de l'opérateur `.*` (appelé pointeur sur *opérateur de sélection de membre*).

Ainsi, si `a` est un objet de classe test, on pourra accéder à la donnée *`i`* de cet objet à travers le pointeur `p1` avec la syntaxe suivante :

```
a.*p1 = 3;  // Initialise la donnée membre i de a avec la valeur 3.
```

Pour les fonctions membres, on mettra des parenthèses à cause des priorités des opérateurs :

```
int i = (a.*p2)();   // Appelle la fonction lit() pour l'objet a.
```

Pour les données et les fonctions membres statiques, cependant, la syntaxe est différente. En effet, les données n'appartiennent plus aux objets de la classe, mais à la classe elle-même, et il n'est plus nécessaire de connaître l'objet auquel le pointeur s'applique pour les utiliser. De même, les fonctions membres statiques ne reçoivent pas le pointeur sur l'objet, et on peut donc les appeler sans référencer ce dernier.

La syntaxe s'en trouve donc modifiée. Les pointeurs sur les membres statiques des classes sont compatibles avec les pointeurs sur les objets et les fonctions non-membres. Par conséquent, si une classe contient une donnée statique entière, on pourra récupérer son adresse directement et la mettre dans un pointeur d'entier :

```
int *p3 = &test::entier_statique;   // Récupère l'adresse
                                    // de la donnée membre
                                    // statique.
```

La même syntaxe s'appliquera pour les fonctions :

```
typedef int (*pg)(void);
pg p4 = &test::fonction_statique;   // Récupère l'adresse
                                    // d'une fonction membre
                                    // statique.
```

Enfin, l'utilisation des ces pointeurs est identique à celle des pointeurs classiques, puisqu'il n'est pas nécessaire de fournir le pointeur `this`. Il est donc impossible de spécifier le pointeur sur l'objet sur lequel la fonction doit travailler aux fonctions membres statiques. Cela est naturel, puisque les fonctions membres statiques ne peuvent pas accéder aux données non statiques d'une classe.

Exemple 7-27. Pointeurs sur membres statiques

```
#include <stdlib.h>
#include <iostream>
using namespace std;

class test
{
    int i;
    static int j;

public:
    test(int j)
    {
        i=j;
        return ;
    }

    static int get(void)
    {
        /* return i ;  INTERDIT : i est non statique
                       et get l'est ! */
        return j; // Autorisé.
    }
```

```
};

int test::j=5;              // Initialise la variable statique.

typedef int (*pf)(void);    // Pointeur de fonction renvoyant
                            // un entier.
pf p=&test::get;            // Initialisation licite, car get
                            // est statique.

int main(void)
{
    cout << (*p)() << endl;// Affiche 5. On ne spécifie pas l'objet.
    return EXIT_SUCCESS;
}
```

Chapitre 8. Les exceptions en C++

Une *exception* est l'interruption de l'exécution du programme à la suite d'un événement particulier. Le but des exceptions est de réaliser des traitements spécifiques aux événements qui en sont la cause. Ces traitements peuvent rétablir le programme dans son mode de fonctionnement normal, auquel cas son exécution reprend. Il se peut aussi que le programme se termine, si aucun traitement n'est approprié.

Le C++ supporte les exceptions logicielles, dont le but est de gérer les erreurs qui surviennent lors de l'exécution des programmes. Lorsqu'une telle erreur survient, le programme doit *lancer une exception*. L'exécution normale du programme s'arrête dès que l'exception est lancée, et le contrôle est passé à un *gestionnaire d'exception*. Lorsqu'un gestionnaire d'exception s'exécute, on dit qu'il a *attrapé l'exception*.

Comme nous allons le voir, les exceptions permettent une gestion simplifiée des erreurs, parce qu'elles en reportent le traitement en dehors de la séquence nominale de l'algorithme, ce qui le simplifie grandement. De plus, elles permettent de régler les problèmes de libération des ressources allouées avant l'apparition de l'erreur de manière automatique, ce qui simplifie d'autant le code de traitement des erreurs. Enfin, les exceptions permettent de spécifier avec précision les erreurs possibles qu'une méthode peut générer, et de classifier les erreurs en catégories d'erreurs sur lesquelles des traitements génériques peuvent être effectuées.

8.1. Techniques de gestion des erreurs

En général, une fonction qui détecte une erreur d'exécution ne peut pas se terminer normalement. Comme son traitement n'a pas pu se dérouler normalement, il est probable que la fonction qui l'a appelée considère elle aussi qu'une erreur a eu lieu et termine son exécution. L'erreur remonte ainsi la liste des appelants de la fonction qui a généré l'erreur. Ce processus continue, de fonction en fonction, jusqu'à ce que l'erreur soit complètement gérée ou jusqu'à ce que le programme se termine (ce cas survient lorsque la fonction principale ne peut pas gérer l'erreur).

Traditionnellement, ce mécanisme est implémenté à l'aide de codes de retour des fonctions. Chaque fonction doit renvoyer une valeur spécifique à l'issue de son exécution, permettant d'indiquer si elle s'est correctement déroulée ou non. La valeur renvoyée est donc utilisée par l'appelant pour déterminer si l'appel s'est bien effectué ou non, et, si erreur il y a, prendre les mesures nécessaires. La nature de l'erreur peut être indiquée soit directement par la valeur retournée par la fonction, soit par une donnée globale que l'appelant peut consulter.

Exemple 8-1. Gestion des erreurs par codes d'erreur

```
// Fonction qui réserve trois ressources et effectue
// un travail avec ces trois ressources.
// Retourne 0 si tout s'est bien passé,
// -1 si la première ressource ne peut être prise,
// -2 si la deuxième ressource ne peut être prise,
// -3 si la troisième ressource ne peut être prie,
// et -1001 à -1004 selon le code d'erreur du travail.
// Les ressources sont consommées par le travail
// si celui-ci réussit.
int fait_boulot(const char *rsrc1, const char *rsrc2,
    const char *rsrc3)
{
    // Initilise le code de résultat à la première
    // erreur possible :
```

```
    int res = -1;
    // Alloue la première ressource :
    int r1 = alloue_ressource(rsrc1);
    if (r1 != 0)
    {
        // Idem avec la deuxième :
        res = -2;
        int r2 = alloue_ressource(rsrc2);
        if (r2 != 0)
        {
            // Idem avec la troisième :
            res = -3;
            int r3 = alloue_ressource(rsrc3);
            if (r3 != 0)
            {
                // OK, on essaie :
                int trv = consomme(r1, r2, r3);
                switch (trv)
                {
                case 0:
                    res = 0;
                    break;
                case 1:
                    res = -1001;
                    break;
                case 2:
                    res = -1002;
                    break;
                case 4:
                    res = -1003;
                    break;
                case 25:
                    res = -1004;
                    break;
                }
                // Il faut libérer en cas d'échec :
                if (res != 0)
                    libere_ressource(r3);
            }
            // Libère r2 :
            if (res != 0)
                libere_ressource(r2);
        }
        if (res != 0)
            libere_ressource(r1);
    }
    return res;
}
```

Malheureusement, comme cet exemple le montre, cette technique nécessite de tester les codes de retour de chaque fonction appelée. La logique d'erreur développée finit par devenir très lourde, puisque ces tests s'imbriquent les uns à la suite des autres et que le code du traitement des erreurs se trouve mélangé avec le code du fonctionnement normal de l'algorithme.

Cette complication peut devenir ingérable lorsque plusieurs valeurs de codes de retour peuvent être renvoyées afin de distinguer les différents cas d'erreurs possibles, car il peut en découler un grand nombre de tests et beaucoup de cas particuliers à gérer dans les fonctions appelantes.

Certains programmes résolvent le problème de l'imbrication des tests d'une manière astucieuse, qui consiste à déporter le traitement des erreurs à effectuer en dehors de l'algorithme par des sauts vers la fin de la fonction. Le code de nettoyage, qui se trouve alors après l'algorithme, est exécuté complètement si tout se passe correctement. En revanche, si la moindre erreur est détectée en cours d'exécution, un saut est réalisé vers la partie du code de nettoyage correspondante au traitement qui a déjà été effectué. Ainsi, ce code n'est écrit qu'une seule fois, et le traitement des erreurs est situé en dehors du traitement normal.

Exemple 8-2. Gestion des erreurs par sauts

```
int fait_boulot(const char *rsrc1, const char *rsrc2,
    const char *rsrc3)
{
    int res;
    // Alloue la première ressource :
    res = -1;
    int r1 = alloue_ressource(rsrc1);
    if (r1 == 0) goto err_alloc_r1;

    // Alloue la deuxième ressource :
    res = -2;
    int r2 = alloue_ressource(rsrc2);
    if (r2 == 0) goto err_alloc_r2;

    // Alloue la troisième ressource :
    res = -3;
    int r3 = alloue_ressource(rsrc3);
    if (r3 == 0) goto err_alloc_r3;

    // Effectue le boulot :
    int trv = consomme(r1, r2, r3);
    switch (trv)
    {
    case 0:
        res = 0;
        goto fin_ok;
    case 1:
        res = -1001;
        break;
    case 2:
        res = -1002;
        break;
    case 4:
        res = -1003;
        break;
    case 25:
        res = -1004;
        break;
    }

err_boulot:
    libere_ressource(r3);
err_alloc_r3:
    libere_ressource(r2);
err_alloc_r2:
    libere_ressource(r1);
```

```
err_alloc_r1:
fin_ok:
    return res;
}
```

La solution précédente est tout à fait valable (en fait, c'est même la solution la plus simple), et elle est très utilisée dans les programmes C bien réalisés. Cependant, elle n'est toujours pas parfaite. En effet, il faut toujours conserver l'information de l'erreur dans le code d'erreur si celle-ci doit être remontée aux fonctions appelantes.

Or, des problèmes fondamentaux sont attachés à la notion de code d'erreur. Premièrement, si l'on veut définir des catégories d'erreurs (par exemple pour les erreurs d'entrée/sortie, les erreurs de droits d'accès, et les erreurs de manque de mémoire), il est nécessaire d'utiliser des plages de valeurs distinctes pour chaque catégorie dans le type utilisé pour les codes d'erreur. Cela nécessite une autorité centrale de définition des codes d'erreurs, faute de quoi des conflits de valeurs pour les code, et donc des interprétations erronnées des erreurs, se produiront. Et cela, croyez moi, ce n'est pas facile du tout à gérer (ce n'est d'ailleurs plus un problème de programmation). Deuxièmement, il n'est a priori pas possible de déterminer tous les codes de retour possibles d'une fonction sans consulter sa documentation ou son code source. Ce problème impose de définir des traitements génériques, alors qu'ils ne sont a posteriori pas nécessaires.

Comme nous allons le voir, la solution qui met en œuvre les exceptions est une alternative intéressante. En effet, la fonction qui détecte une erreur peut se contenter de lancer une exception en lui fournissant le contexte de l'erreur. Cette exception interrompt l'exécution de la fonction, et un gestionnaire d'exception approprié est recherché. La recherche du gestionnaire suit le même chemin que celui utilisé lors de la remontée classique des erreurs : à savoir la liste des appelants. Le premier bloc d'instructions qui contient un gestionnaire d'exception capable de traiter cette exception prend donc le contrôle, et effectue le traitement de l'erreur. Si le traitement est complet, le programme reprend son exécution normale. Sinon, le gestionnaire d'exception peut relancer l'exception (auquel cas le gestionnaire d'exception suivant pour ce type d'exception est recherché) ou terminer le programme.

Les gestionnaires d'exceptions capables de traiter une exception sont identifiés par les mécanismes de typage du langage. Ainsi, plusieurs types d'exceptions, a priori définis de manière indépendants, peuvent être utilisés. De plus, le polymorphisme des exceptions permet de les structurer facilement afin de prendre en charge des catégories d'erreurs. Enfin, toutes les informations relatives à l'erreur, et notamment son contexte, sont transmises aux gestionnaires d'exceptions automatiquement par le mécanisme des exceptions du langage. Ainsi, il n'y a plus de risque de conflit lors de la définition de codes d'erreurs, ni de données globales utilisées pour stocker les informations descriptives de l'erreur. Pour finir, chaque fonction peut spécifier les exceptions qu'elle peut lancer, ce qui permet à l'appelant de savoir à quoi s'attendre.

Les exceptions permettent donc de simplifier le code et de le rendre plus fiable. Par ailleurs, la logique d'erreur est complètement prise en charge par le langage.

> **Note :** L'utilisation des exceptions n'est pour autant pas forcément la meilleure des choses dans un programme. En effet, si un programme utilise du code qui gère les exceptions et du code qui ne gère pas les exceptions (par exemple dans deux bibliothèques tierces), il aura la lourde tâche soit d'encapsuler le code qui ne gère pas les exceptions pour pouvoir l'utiliser correctement, soit de prendre en charge les deux types de traitements d'erreurs directement. La première solution est très lourde et coûteuse, et est difficilement justifiable par des considérations de facilité de traitement des erreurs. La deuxième solution est encore pire, le code devenant absolument horrible.
>
> Dans ce genre de situations, mieux vaut s'adapter et utiliser le mécanisme majoritaire. Très souvent hélas, il faut abandonner les exceptions et utiliser les notions de codes d'erreur.

Nous allons à présent voir comment utiliser les exceptions en C++.

8.2. Lancement et récupération d'une exception

En C++, lorsqu'il faut lancer une exception, on doit créer un objet dont la classe caractérise cette exception, et utiliser le mot clé `throw`. Sa syntaxe est la suivante :

```
throw objet;
```

où `objet` est l'objet correspondant à l'exception. Cet objet peut être de n'importe quel type, et pourra ainsi caractériser pleinement l'exception.

Les exceptions doivent alors être traitées par un gestionnaire d'exception qui leur correspond. Pour cela, il faut délimiter chaque zone de code susceptible de lancer des exceptions. Cela se fait en plaçant le code à protéger dans un bloc d'instructions particulier. Ce bloc est introduit avec le mot clé `try` :

```
try
{
    // Code susceptible de générer des exceptions...
}
```

Les gestionnaires d'exceptions doivent suivre le bloc `try`. Ils sont introduits avec le mot clé `catch` :

```
catch (classe [&][temp])
{
    // Traitement de l'exception associée à la classe
}
```

Dès qu'une exception est lancée, le compilateur recherche un gestionnaire d'exception approprié en remontant les blocs d'instructions et la pile d'appel des fonctions. À chaque étape, les objets de classe de stockage automatique définis dans les blocs dont la remontée de l'exception en fait sortir le contrôle du programme sont bien entendu automatiquement détruits. De ce fait, si l'ensemble des ressources utilisées par le programme est encapsulé dans des classes dont les destructeurs sont capables de les détruire ou de les ramener dans un état cohérent, la gestion des ressources devient totalement automatique pendant les traitements d'erreurs.

Les traitements que l'on doit effectuer dans les blocs `catch` sont les traitements d'erreurs que le C++ ne fera pas automatiquement. Ces traitements comprennent généralement le rétablissement de l'état des données manipulées par le code qui a lancé l'exception (dont, pour les fonctions membres d'une classe, les données membres de l'objet courant), ainsi que la libération des ressources non encapsulées dans des objets de classe de stockage automatique (par exemple, les fichiers ouverts, les connexions réseau, etc.).

Un gestionnaire d'exception peut relancer l'exception s'il le désire, par exemple pour permettre aux gestionnaires de niveau supérieur de faire la suite du traitement d'erreur. Pour cela, il suffit d'utiliser le mot clé `throw`. La syntaxe est la suivante :

```
throw ;
```

L'exception est alors relancée, et un nouveau bloc `catch` est recherché avec les mêmes paramètres. Le parcours de l'exception s'arrêtera donc dès que l'erreur aura été complètement traitée.

> **Note :** Bien entendu, il est possible de lancer une autre exception que celle que l'on a reçue, comme ce peut être par exemple le cas si le traitement de l'erreur provoque lui-même une erreur.

Il peut y avoir plusieurs gestionnaires d'exceptions. Chacun traitera les exceptions qui ont été générées dans le bloc `try` et dont l'objet est de la classe indiquée par son paramètre. Il n'est pas nécessaire de donner un nom à l'objet (`temp`) dans l'expression `catch`. Cependant, cela permet de le récupérer, ce qui peut être nécessaire si l'on doit récupérer des informations sur la nature de l'erreur.

Enfin, il est possible de définir un gestionnaire d'exception universel, qui récupérera toutes les exceptions possibles, quels que soient leurs types. Ce gestionnaire d'exception doit prendre comme paramètre trois points de suspension entre parenthèses dans sa clause `catch`. Bien entendu, dans ce cas, il est impossible de spécifier une variable qui contient l'exception, puisque son type est indéfini.

Exemple 8-3. Utilisation des exceptions

```
#include <stdlib.h>
#include <iostream>
using namespace std;

class erreur   // Première exception possible, associée
               // à l'objet erreur.
{
public:
    int cause;  // Entier spécifiant la cause de l'exception.
    // Le constructeur. Il appelle le constructeur de cause.
    erreur(int c) : cause(c) {}
    // Le constructeur de copie. Il est utilisé par le mécanisme
    // des exceptions :
    erreur(const erreur &source) : cause(source.cause) {}
};

class other {};   // Objet correspondant à toutes
                  // les autres exceptions.

int main(void)
{
    int i;            // Type de l'exception à générer.
    cout << "Tapez 0 pour générer une exception Erreur, "
        "1 pour une Entière :";
    cin >> i;         // On va générer une des trois exceptions
                      // possibles.
    cout << endl;
    try               // Bloc où les exceptions sont prises en charge.
    {
        switch (i)    // Selon le type d'exception désirée,
        {
        case 0:
            {
                erreur a(0);
                throw (a);   // on lance l'objet correspondant
                             // (ici, de classe erreur).
                             // Cela interrompt le code. break est
                             // donc inutile ici.
            }
        case 1:
```

```
            {
                int a=1;
                throw (a);   // Exception de type entier.
            }
        default:             // Si l'utilisateur n'a pas tapé 0 ou 1,
            {
                other c;     // on crée l'objet c (type d'exception
                throw (c);   // other) et on le lance.
            }
        }
    }                 // fin du bloc try. Les blocs catch suivent :
    catch (erreur &tmp) // Traitement de l'exception erreur ...
    {                 // (avec récupération de la cause).
        cout << "Erreur erreur ! (cause " << tmp.cause << ")" << endl;
    }
    catch (int tmp)   // Traitement de l'exception int...
    {
        cout << "Erreur int ! (cause " << tmp << ")" << endl;
    }
    catch (...)       // Traitement de toutes les autres
    {                 // exceptions (...).
                      // On ne peut pas récupérer l'objet ici.
        cout << "Exception inattendue !" << endl;
    }
    return EXIT_SUCCESS;
}
```

Selon ce qu'entre l'utilisateur, une exception du type erreur, int ou other est générée.

8.3. Hiérarchie des exceptions

Le mécanisme des exceptions du C++ se base sur le typage des objets, puisque le lancement d'une exception nécessite la construction d'un objet qui la caractérise, et le bloc `catch` destination de cette exception sera sélectionné en fonction du type de cet objet. Bien entendu, les objets utilisés pour lancer les exceptions peuvent contenir des informations concernant la nature des erreurs qui se produisent, mais il est également possible de classifier ces erreurs par catégories en se basant sur leurs types.

En effet, les objets exceptions peuvent être des instances de classes disposant de relations d'héritage. Comme les objets des classes dérivées peuvent être considérés comme des instances de leurs classes de base, les gestionnaires d'exception peuvent récupérer les exceptions de ces classes dérivées en récupérant un objet du type d'une de leurs classes de base. Ainsi, il est possible de classifier les différents cas d'erreurs en définissant une hiérarchie de classe d'exceptions, et d'écrire des traitements génériques en n'utilisant que les objets d'un certain niveau dans cette hiérarchie.

Le mécanisme des exceptions se montre donc plus puissant que toutes les autres méthodes de traitement d'erreurs à ce niveau, puisque la sélection du gestionnaire d'erreur est automatiquement réalisée par le langage. Cela peut être très pratique pour peu que l'on ait défini correctement sa hiérarchie de classes d'exceptions.

Exemple 8-4. Classification des exceptions

```
#include <stdlib.h>
#include <iostream>
using namespace std;
```

```
// Classe de base de toutes les exceptions :
class ExRuntimeError
{
};

// Classe de base des exceptions pouvant se produire
// lors de manipulations de fichiers :
class ExFileError : public ExRuntimeError
{
};

// Classes des erreurs de manipulation des fichiers :
class ExInvalidName : public ExFileError
{
};

class ExEndOfFile : public ExFileError
{
};

class ExNoSpace : public ExFileError
{
};

class ExMediumFull : public ExNoSpace
{
};

class ExFileSizeMaxLimit : public ExNoSpace
{
};

// Fonction faisant un travail quelconque sur un fichier :
void WriteData(const char *szFileName)
{
    // Exemple d'erreur :
    if (szFileName == NULL) throw ExInvalidName();
    else
    {
        // Traitement de la fonction
        // etc.

        // Lancement d'une exception :
        throw ExMediumFull();
    }
}

void Save(const char *szFileName)
{
    try
    {
        WriteData(szFileName);
    }
    // Traitement d'un erreur spécifique :
    catch (ExInvalidName &)
    {
```

```
        cout << "Impossible de faire la sauvegarde" << endl;
    }
    // Traitement de toutes les autres erreurs en groupe :
    catch (ExFileError &)
    {
        cout << "Erreur d'entrée / sortie" << endl;
    }
}

int main(void)
{
    Save(NULL);
    Save("data.dat");
    return EXIT_SUCCESS;
}
```

La bibliothèque standard C++ définit elle-même un certain nombre d'exceptions standards, qui sont utilisées pour signaler les erreurs qui se produisent à l'exécution des programmes. Quelques-unes de ces exceptions ont déjà été présentées avec les fonctionnalités qui sont susceptibles de les lancer. Vous trouverez une liste complète des exceptions de la bibliothèque standard du C++ dans la Section 13.2.

8.4. Traitement des exceptions non captées

Si, lorsqu'une exception se produit dans un bloc `try`, il est impossible de trouver le bloc `catch` correspondant à la classe de cette exception, il se produit une erreur d'exécution. La fonction prédéfinie `std::terminate` est alors appelée. Elle se contente d'appeler une fonction de traitement de l'erreur, qui elle-même appelle la fonction `abort` de la bibliothèque C. Cette fonction termine en catastrophe l'exécution du programme fautif en générant une faute (les ressources allouées par le programme ne sont donc pas libérées, et des données peuvent être perdues). Ce n'est généralement pas le comportement désiré, aussi est-il possible de le modifier en changeant la fonction appelée par `std::terminate`.

Pour cela, il faut utiliser la fonction `std::set_terminate`, qui attend en paramètre un pointeur sur la fonction de traitement d'erreur, qui ne prend aucun paramètre et renvoie void. La valeur renvoyée par `std::set_terminate` est le pointeur sur la fonction de traitement d'erreur précédente. `std::terminate` et `std::set_terminate` sont déclaréee dans le fichier d'en-tête `exception`.

> **Note :** Comme leurs noms l'indiquent, `std::terminate` et `std::set_terminate` sont déclarées dans l'espace de nommage `std::`, qui est réservé pour tous les objets de la bibliothèque standard C++. Si vous ne voulez pas à avoir à utiliser systématiquement le préfixe `std::` devant ces noms, vous devrez ajouter la ligne « `using namespace std;` » après avoir inclus l'en-tête `exception`. Vous obtiendrez de plus amples renseignements sur les espaces de nommage dans le Chapitre 10.

Exemple 8-5. Installation d'un gestionnaire d'exception avec set_terminate

```
#include <stdlib.h>
#include <iostream>
#include <exception>
using namespace std;

void mon_gestionnaire(void)
```

```
{
    cout << "Exception non gérée reçue !" << endl;
    cout << "Je termine le programme proprement..."
         << endl;
    exit(-1);
}

int lance_exception(void)
{
    throw 2;
}

int main(void)
{
    set_terminate(&mon_gestionnaire);
    try
    {
        lance_exception();
    }
    catch (double d)
    {
        cout << "Exception de type double reçue : " <<
           d << endl;
    }
    return EXIT_SUCCESS;
}
```

8.5. Liste des exceptions autorisées pour une fonction

Il est possible de spécifier les exceptions qui peuvent être lancées par une fonction. Pour cela, il faut faire suivre son en-tête du mot clé `throw` avec, entre parenthèses et séparées par des virgules, les classes des exceptions qu'elle est autorisée à lancer. Par exemple, la fonction suivante :

```
int fonction_sensible(void)
    throw (int, double, erreur)
{
    ...
}
```

n'a le droit de lancer que des exceptions du type int, double ou erreur. Si une exception d'un autre type est lancée, par exemple une exception du type char *, il se produit encore une fois une erreur à l'exécution.

Dans ce cas, la fonction `std::unexpected` est appelée. Cette fonction se comporte de manière similaire à `std::terminate`, puisqu'elle appelle par défaut une fonction de traitement de l'erreur qui elle-même appelle la fonction `std::terminate` (et donc `abort` en fin de compte). Cela conduit à la terminaison du programme. On peut encore une fois changer ce comportement par défaut en remplaçant la fonction appelée par `std::unexpected` par une autre fonction à l'aide de `std::set_unexpected`, qui est déclarée dans le fichier d'en-tête `exception`. Cette dernière attend en paramètre un pointeur sur la fonction de traitement d'erreur, qui ne doit prendre aucun paramètre et qui ne doit rien renvoyer. `std::set_unexpected` renvoie le pointeur sur la fonction de traitement d'erreur précédemment appelée par `std::unexpected`.

> **Note :** Comme leurs noms l'indiquent, `std::unexpected` et `std::set_unexpected` sont déclarées dans l'espace de nommage `std::`, qui est réservé pour les objets de la bibliothèque standard C++. Si vous ne voulez pas avoir à utiliser systématiquement le préfixe `std::` pour ces noms, vous devrez ajouter la ligne « `using namespace std;` » après avoir inclus l'en-tête `exception`. Vous obtiendrez de plus amples renseignements sur les espaces de nommage dans le Chapitre 10.

Il est possible de relancer une autre exception à l'intérieur de la fonction de traitement d'erreur. Si cette exception satisfait la liste des exceptions autorisées, le programme reprend son cours normalement dans le gestionnaire correspondant. C'est généralement ce que l'on cherche à faire. Le gestionnaire peut également lancer une exception de type std::bad_exception, déclarée comme suit dans le fichier d'en-tête `exception` :

```
class bad_exception : public exception
{
public:
    bad_exception(void) throw();
    bad_exception(const bad_exception &) throw();
    bad_exception &operator=(const bad_exception &) throw();
    virtual ~bad_exception(void) throw();
    virtual const char *what(void) const throw();
};
```

Cela a pour conséquence de terminer le programme.

Enfin, le gestionnaire d'exceptions non autorisées peut directement mettre fin à l'exécution du programme en appelant `std::terminate`. C'est le comportement utilisé par la fonction `std::unexpected` définie par défaut.

Exemple 8-6. Gestion de la liste des exceptions autorisées

```
#include <stdlib.h>
#include <iostream>
#include <exception>
using namespace std;

void mon_gestionnaire(void)
{
    cout << "Une exception illégale a été lancée." << endl;
    cout << "Je relance une exception de type int." << endl;
    throw 2;
}

int f(void) throw (int)
{
    throw "5.35";
}

int main(void)
{
    set_unexpected(&mon_gestionnaire);
    try
    {
        f();
    }
```

```
    catch (int i)
    {
        cout << "Exception de type int reçue : " <<
           i << endl;
    }
    return EXIT_SUCCESS;
}
```

Note : La liste des exceptions autorisées dans une fonction ne fait pas partie de sa signature. Elle n'intervient donc pas dans les mécanismes de surcharge des fonctions. De plus, elle doit se placer après le mot clé `const` dans les déclarations de fonctions membres `const` (en revanche, elle doit se placer avant `=0` dans les déclarations des fonctions virtuelles pures).

On prendra garde au fait que les exceptions ne sont pas générées par le mécanisme de gestion des erreurs du C++ (ni du C). Cela signifie que pour avoir une exception, il faut la lancer, le compilateur ne fera pas les tests pour vous (tests de débordements numériques dans les calculs par exemple). Cela supposerait de prédéfinir un ensemble de classes pour les erreurs génériques. Les tests de validité d'une opération doivent donc être faits malgré tout et, le cas échéant, il faut lancer une exception pour reporter le traitement en cas d'échec. De même, les exceptions générées par la machine hôte du programme ne sont en général pas récupérées par les implémentations et, si elles le sont, les programmes qui les utilisent ne sont pas portables.

8.6. Gestion des objets exception

Lors de la lancée d'une exception, l'objet exception fourni en paramètre à `throw` peut être copié dans une variable temporaire prise en charge par le compilateur ou non. Dans ce dernier cas, l'objet exception est construit directement dans cette variable temporaire, et utilisé directement pour propager l'exception. Le comportement exact n'est pas fixé par la norme, et relève d'une optimisation.

Note : Cela implique que les programmes portables doivent absolument éviter les effets de bord dans le traitement des constructeurs de copie et des destructeurs des classes utilisées pour les exceptions.

L'objet temporaire utilisé par le compilateur pour propager l'exception est utilisé pour initialiser les blocs `catch` qui la traitent. L'utilisation de cet objet par les blocs `catch` se fait exactement de la même manière que pour le passage d'une variable en paramètre à une fonction. Les blocs `catch` peuvent donc recevoir leurs paramètres par valeur ou par référence. Lorsque l'objet exception est reçu par valeur, une copie supplémentaire est réalisée. Lorsqu'il est reçu par référence, cette copie est évitée, mais toutes les modifications effectuées sur l'objet exception seront effectuées dans la copie de travail du compilateur, et seront donc également visibles dans les blocs `catch` des fonctions appelantes ou de portée supérieure si l'exception est relancée après traitement.

Afin d'éviter les copies d'objets inutiles, il est recommandé de capter les exceptions par référence. On pourra utiliser le mot clé `const` pour éviter les effets de bords si ceux-ci ne sont pas désirés.

L'objet temporaire utilisé par le compilateur pour propager l'exception est détruit automatiquement une fois que le traitement d'exception est terminé.

Pendant tout le traitement de propagation d'une exception, y compris pendant la destruction automatique des objets dont la portée est quittée, la fonction globale `std::uncaught_exception` renvoie

`true`. Cette fonction est déclarée dans le fichier d'en-tête `exception` comme une fonction retournant un booléen et ne prenant aucun paramètre. Cette fonction renvoie `false` dès que l'exception est attrapée par l'une des clauses `catch`, ou dès qu'il n'y a plus de traitement d'exception en cours.

8.7. Exceptions dans les constructeurs et les destructeurs

Il est parfaitement légal de lancer une exception dans un constructeur. En fait, c'est même la seule solution pour signaler une erreur lors de la construction d'un objet, puisque les constructeurs n'ont pas de valeur de retour.

Lorsqu'une exception est lancée à partir d'un constructeur, la construction de l'objet échoue. Par conséquent, le compilateur n'appellera jamais le destructeur pour cet objet, puisque cela n'a pas de sens. En revanche, les données membres et les classes de bases déjà contruites sont automatiquement détruites avant de remonter l'exception. De ce fait, toutes les ressources que le constructeur a commencé à réserver sont libérées automatiquement, pourvu qu'elles soient encapsulées dans des classes disposant de destructeurs.

Pour les autres ressources (ressources systèmes ou pointeurs contenant des données allouées dynamiquement), aucun code de destruction n'est exécuté. Cela peut conduire à des fuites de mémoire ou à une consommation de ressource qui ne pourront plus être libérées. De ce fait, il est conseillé de ne jamais faire de traitements complexes ou susceptibles de lancer une exception dans un constructeur, et de définir une fonction d'initialisation que l'on appelera dans un contexte plus fiable. Si cela n'est pas possible, il est impératif d'encapsuler toutes les ressources que le constructeur réservera dans des classes disposant d'un destructeur afin de libérer correctement la mémoire.

De même, lorsque la construction de l'objet se fait dans le cadre d'une allocation dynamique de mémoire, le compilateur appelle automatiquement l'opérateur `delete` afin de restituer la mémoire allouée pour cet objet. Il est donc inutile de restituer la mémoire de l'objet alloué dans le traitement de l'exception qui suit la création dynamique de l'objet, et il ne faut pas y appeler l'opérateur `delete` manuellement.

> **Note :** Le compilateur détruit les données membres et les classes de base qui ont été construites avant le lancement de l'exception avant d'appeler l'opérateur delete pour libérer la mémoire. D'ordinaire, c'est cet opérateur qui appelle le destructeur de l'objet à détruire, mais il ne le fait pas une deuxième fois. Le comportement de l'opérateur `delete` est donc lui aussi légèrement modifié par le mécanisme des exceptions lorsqu'il s'en produit une pendant la construction d'un objet.

Il est possible de capter les exceptions qui se produisent pendant l'exécution du constructeur d'une classe, afin de faire un traitement sur cette exception, ou éventuellement afin de la traduire dans une autre exception. Pour cela, le C++ fournit une syntaxe particulière. Cette syntaxe permet simplement d'utiliser un bloc `try` pour le corps de fonction des constructeurs. Les blocs `catch` suivent alors la définition du constructeur, et effectuent les traitements sur l'exception.

Exemple 8-7. Exceptions dans les constructeurs

```
#include <stdlib.h>
#include <iostream>
using namespace std;
```

```
class A
{
    struct Buffer
    {
        char *p;

        Buffer() : p(0)
        {
        }

        ~Buffer()
        {
            if (p != 0)
                delete[] p;
        }
    };

    Buffer pBuffer;             // Pointeur à libération automatique.
    int  *pData;                // Pointeur classique.

public:
    A() throw (double);
    ~A();

    static void *operator new(size_t taille)
    {
        cout << "new" << endl;
        return malloc(taille);
    }

    static void operator delete(void *p)
    {
        cout << "delete" << endl;
        free(p);
    }
};

// Constructeur susceptible de lancer une exception :
A::A() throw (double)
try : pData()
{
    cout << "Début du constructeur" << endl;
    pBuffer.p = new char[10];
    pData = new int[10];        // Fuite de mémoire !
    cout << "Lancement de l'exception" << endl;
    throw 2;
}
catch (int)
{
    cout << "catch du constructeur..." << endl;
    // delete[] pData;          // Illégal! L'objet est déjà détruit !
    // Conversion de l'exception en flottant :
    throw 3.14;
}

A::~A()
{
```

```
    // Libération des données non automatiques :
    delete[] pData;             // Jamais appelé !
    cout << "A::~A()" << endl;
}

int main(void)
{
    try
    {
        A *a = new A;
    }
    catch (double d)
    {
        cout << "Aïe, même pas mal !" << endl;
    }
    return EXIT_SUCCESS;
}
```

Dans cet exemple, lors de la création dynamique d'un objet A, une erreur d'initialisation se produit et une exception de type entier est lancée. Celle-ci est alors traitée dans le bloc `catch` qui suit la définition du constructeur de la classe A. Ce bloc convertit l'exception en exception de type flottant. L'opérateur `delete` est bien appelé automatiquement, mais le destructeur de A n'est jamais exécuté. Comme indiqué dans le bloc `catch` du constructeur, les données membres de l'objet, qui est déjà détruit, ne sont plus accessibles. De ce fait, le bloc alloué dynamiquement pour `pData` ne peut être libéré, et comme le destructeur n'est pas appelé, ce bloc de mémoire est définitivement perdu. On ne peut corriger le problème que de deux manières : soit on n'effectue pas ce type de traitement dans le constructeur, soit on encapsule le pointeur dans une classe disposant d'un destructeur. L'exemple précédent définit une classe Buffer pour la donnée membre `pBuffer`. Pour information, la bibliothèque standard C++ fournit une classe générique similaire que l'on pourra également utiliser dans ce genre de situation (voir la Section 14.2.1 pour plus de détails à ce sujet).

Le comportement du bloc `catch` des constructeurs avec bloc `try` est différent de celui des blocs `catch` classiques. L'objet n'ayant pu être construit, l'exception doit obligatoirement être transmise au code qui a cherché à le créer, afin d'éviter de le laisser dans un état incohérent (c'est-à-dire avec une référence d'objet non construit). Par conséquent, contrairement aux exceptions traitées dans un bloc `catch` classique, les exceptions lancées dans les constructeurs sont automatiquement relancées une fois qu'elles ont été traitées dans le bloc `catch` du constructeur. L'exception doit donc toujours être captée dans le code qui cherche à créer l'objet, afin de prendre les mesures adéquates. Ainsi, dans l'exemple précédent, si l'exception n'était pas convertie en exception de type flottant, le programme planterait, car elle serait malgré tout relancée automatiquement et la fonction `main` ne dispose pas de bloc `catch` pour les exceptions de type entier.

> **Note :** Cette règle implique que les programmes déclarant des objets globaux dont le constructeur peut lancer une exception risquent de se terminer en catastrophe. En effet, si une exception est lancée par ce constructeur à l'initialisation du programme, aucun gestionnaire d'exception ne sera en mesure de la capter lorsque le bloc `catch` la relancera. On évitera donc à tout prix de créer des objets globaux dont les constructeurs effectuent des tâches susceptibles de lancer une exception. De manière générale, comme il l'a déjà été dit, il n'est de toutes manières pas sage de réaliser des tâches complexes dans un constructeur et lors de l'initialisation des données statiques.

En général, si une classe hérite de une ou plusieurs classes de base, l'appel aux constructeurs des classes de base doit se faire entre le mot clé `try` et la première accolade. En effet, les constructeurs

des classes de base sont susceptibles, eux aussi, de lancer des exceptions. La syntaxe est alors la suivante :

```
Classe::Classe
try : Base(paramètres) [, Base(paramètres) [...]]
{
}
catch ...
```

Enfin, les exceptions lancées dans les destructeurs ne peuvent être captées. En effet, elles correspondent à une situation grave, puisqu'il n'est pas pas possible de détruire correctement l'objet dans ce cas. Dans ce cas, la fonction `std::terminate` est appelée automatiquement, et le programme se termine en conséquence.

Chapitre 9. Identification dynamique des types

Le C++ est un langage fortement typé. Malgré cela, il se peut que le type exact d'un objet soit inconnu à cause de l'héritage. Par exemple, si un objet est considéré comme un objet d'une classe de base de sa véritable classe, on ne peut pas déterminer a priori quelle est sa véritable nature.

Cependant, les objets polymorphiques (qui, rappelons-le, sont des objets disposant de méthodes virtuelles) conservent des informations sur leur *type dynamique*, à savoir leur véritable nature. En effet, lors de l'appel des méthodes virtuelles, la méthode appelée est la méthode de la véritable classe de l'objet.

Il est possible d'utiliser cette propriété pour mettre en place un mécanisme permettant d'identifier le type dynamique des objets, mais cette manière de procéder n'est pas portable. Le C++ fournit donc un mécanisme standard permettant de manipuler les informations de type des objets polymorphiques. Ce mécanisme prend en charge l'*identification dynamique* des types et la *vérification de la validité des transtypages* dans le cadre de la dérivation.

9.1. Identification dynamique des types

9.1.1. L'opérateur typeid

Le C++ fournit l'opérateur `typeid`, qui permet de récupérer les informations de type des expressions. Sa syntaxe est la suivante :

```
typeid(expression)
```

où `expression` est l'expression dont il faut déterminer le type.

Le résultat de l'opérateur `typeid` est une référence sur un objet constant de classe type_info. Cette classe sera décrite dans la Section 9.1.2.

Les informations de type récupérées sont les informations de type statique pour les types non polymorphiques. Cela signifie que l'objet renvoyé par `typeid` caractérisera le type de l'expression fournie en paramètre, que cette expression soit un sous-objet d'un objet plus dérivé ou non. En revanche, pour les types polymorphiques, si le type ne peut pas être déterminé statiquement (c'est-à-dire à la compilation), une détermination dynamique (c'est-à-dire à l'exécution) du type a lieu, et l'objet de classe type_info renvoyé décrit le vrai type de l'expression (même si elle représente un sous-objet d'un objet d'une classe dérivée). Cette situation peut arriver lorsqu'on manipule un objet à l'aide d'un pointeur ou d'une référence sur une classe de base de la classe de cet objet.

Exemple 9-1. Opérateur typeid

```
#include <stdlib.h>
#include <typeinfo>
using namespace std;

class Base
{
public:
    virtual ~Base(void);   // Il faut une fonction virtuelle
                           // pour avoir du polymorphisme.
```

```
};

Base::~Base(void)
{
    return ;
}

class Derivee : public Base
{
public:
    virtual ~Derivee(void);
};

Derivee::~Derivee(void)
{
    return ;
}

int main(void)
{
    Derivee* pd = new Derivee;
    Base* pb = pd;
    const type_info &t1=typeid(*pd);  // t1 qualifie le type de *pd.
    const type_info &t2=typeid(*pb);  // t2 qualifie le type de *pb.
    return EXIT_SUCCESS;
}
```

Les objets `t1` et `t2` sont égaux, puisqu'ils qualifient tous les deux le même type (à savoir, la classe Derivee). `t2` ne contient pas les informations de type de la classe Base, parce que le vrai type de l'objet pointé par `pb` est la classe Derivee.

> **Note :** Notez que la classe type_info est définie dans l'espace de nommage `std::`, réservé à la bibliothèque standard C++, dans l'en-tête `typeinfo`. Par conséquent, son nom doit être précédé du préfixe `std::`. Vous pouvez vous passer de ce préfixe en important les définitions de l'espace de nommage de la bibliothèque standard à l'aide d'une directive `using`. Vous trouverez de plus amples renseignements sur les espaces de nommage dans le Chapitre 10.

On fera bien attention à déréférencer les pointeurs, car sinon, on obtient les informations de type sur ce pointeur, pas sur l'objet pointé. Si le pointeur déréférencé est le pointeur nul, l'opérateur `typeid` lance une exception dont l'objet est une instance de la classe bad_typeid. Cette classe est définie comme suit dans l'en-tête `typeinfo` :

```
class bad_typeid : public logic
{
public:
    bad_typeid(const char * what_arg) : logic(what_arg)
    {
        return ;
    }

    void raise(void)
    {
        handle_raise();
        throw *this;
    }
```

```
};
```

9.1.2. La classe type_info

Les informations de type sont enregistrées dans des objets de la classe type_info prédéfinie par le langage. Cette classe est déclarée dans l'en-tête `typeinfo` de la manière suivante :

```
class type_info
{
public:
    virtual ~type_info();
    bool operator==(const type_info &rhs) const;
    bool operator!=(const type_info &rhs) const;
    bool before(const type_info &rhs) const;
    const char *name() const;
private:
    type_info(const type_info &rhs);
    type_info &operator=(const type_info &rhs);
};
```

Les objets de la classe type_info ne peuvent pas être copiés, puisque l'opérateur d'affectation et le constructeur de copie sont tous les deux déclarés `private`. Par conséquent, le seul moyen de générer un objet de la classe type_info est d'utiliser l'opérateur `typeid`.

Les opérateurs de comparaison permettent de tester l'égalité et la différence de deux objets type_info, ce qui revient exactement à comparer les types des expressions.

Les objets type_info contiennent des informations sur les types sous la forme de chaînes de caractères. Une de ces chaînes représente le type sous une forme lisible par un être humain, et une autre sous une forme plus appropriée pour le traitement des types. Le format de ces chaînes de caractères n'est pas précisé et peut varier d'une implémentation à une autre. Il est possible de récupérer le nom lisible du type à l'aide de la méthode `name`. La valeur renvoyée est un pointeur sur une chaîne de caractères. On ne doit pas libérer la mémoire utilisée pour stocker cette chaîne de caractères.

La méthode `before` permet de déterminer un ordre dans les différents types appartenant à la même hiérarchie de classes, en se basant sur les propriétés d'héritage. L'utilisation de cette méthode est toutefois difficile, puisque l'ordre entre les différentes classes n'est pas fixé et peut dépendre de l'implémentation.

9.2. Transtypages C++

Les règles de dérivation permettent d'assurer le fait que lorsqu'on utilise un pointeur sur une classe, l'objet pointé existe bien et est bien de la classe sur laquelle le pointeur est basé. En particulier, il est possible de convertir un pointeur sur un objet en un pointeur sur un sous-objet.

En revanche, il est interdit d'utiliser un pointeur sur une classe de base pour initialiser un pointeur sur une classe dérivée. Pourtant, cette opération peut être légale, si le programmeur sait que le pointeur pointe bien sur un objet de la classe dérivée. Le langage exige cependant un transtypage explicite. Une telle situation demande l'analyse du programme afin de savoir si elle est légale ou non.

Parfois, il est impossible de faire cette analyse. Cela signifie que le programmeur ne peut pas certifier que le pointeur dont il dispose est un pointeur sur un sous-objet. Le mécanisme d'identification dynamique des types peut être alors utilisé pour vérifier, à l'exécution, si le transtypage est légal. S'il ne l'est pas, un traitement particulier doit être effectué, mais s'il l'est, le programme peut se poursuivre normalement.

Le C++ fournit un jeu d'opérateurs de transtypage qui permettent de faire ces vérifications dynamiques, et qui donc sont nettement plus sûrs que le transtypage tout puissant du C que l'on a utilisé jusqu'ici. Ces opérateurs sont capables de faire un *transtypage dynamique*, un *transtypage statique*, un *transtypage de constance* et un *transtypage de réinterprétation* des données. Nous allons voir les différents opérateurs permettant de faire ces transtypages, ainsi que leur signification.

9.2.1. Transtypage dynamique

Le *transtypage dynamique* permet de convertir une expression en un pointeur ou une référence d'une classe, ou un pointeur sur void. Il est réalisé à l'aide de l'opérateur `dynamic_cast`. Cet opérateur impose des restrictions lors des transtypages afin de garantir une plus grande fiabilité :

- il effectue une vérification de la validité du transtypage ;
- il n'est pas possible d'éliminer les qualifications de constance (pour cela, il faut utiliser l'opérateur `const_cast`, que l'on verra plus loin).

En revanche, l'opérateur `dynamic_cast` permet parfaitement d'accroître la constance d'un type complexe, comme le font les conversions implicites du langage vues dans la Section 3.6 et dans la Section 4.7.

Il ne peut pas travailler sur les types de base du langage, sauf void *.

La syntaxe de l'opérateur `dynamic_cast` est donnée ci-dessous :

```
dynamic_cast<type>(expression)
```

où `type` désigne le type cible du transtypage, et `expression` l'expression à transtyper.

Le transtypage d'un pointeur ou d'une référence d'une classe dérivée en classe de base se fait donc directement, sans vérification dynamique, puisque cette opération est toujours valide. Les lignes suivantes :

```
// La classe B hérite de la classe A :

B *pb;
A *pA=dynamic_cast<A *>(pB);
```

sont donc strictement équivalentes à celles-ci :

```
// La classe B hérite de la classe A :

B *pb;
A *pA=pB;
```

Tout autre transtypage doit se faire à partir d'un type polymorphique, afin que le compilateur puisse utiliser l'identification dynamique des types lors du transtypage. Le transtypage d'un pointeur d'un objet vers un pointeur de type void renvoie l'adresse du début de l'objet le plus dérivé, c'est-à-dire

l'adresse de l'objet complet. Le transtypage d'un pointeur ou d'une référence sur un sous-objet d'un objet vers un pointeur ou une référence de l'objet complet est effectué après vérification du type dynamique. Si l'objet pointé ou référencé est bien du type indiqué pour le transtypage, l'opération se déroule correctement. En revanche, s'il n'est pas du bon type, `dynamic_cast` n'effectue pas le transtypage. Si le type cible est un pointeur, le pointeur nul est renvoyé. Si en revanche l'expression caractérise un objet ou une référence d'objet, une exception de type bad_cast est lancée.

La classe bad_cast est définie comme suit dans l'en-tête `typeinfo` :

```
class bad_cast : public exception
{
public:
    bad_cast(void) throw();
    bad_cast(const bad_cast&) throw();
    bad_cast &operator=(const bad_cast&) throw();
    virtual ~bad_cast(void) throw();
    virtual const char* what(void) const throw();
};
```

Lors d'un transtypage, aucune ambiguïté ne doit avoir lieu pendant la recherche dynamique du type. De telles ambiguïtés peuvent apparaître dans les cas d'héritage multiple, où plusieurs objets de même type peuvent coexister dans le même objet. Cette restriction mise à part, l'opérateur `dynamic_cast` est capable de parcourir une hiérarchie de classe aussi bien verticalement (convertir un pointeur de sous-objet vers un pointeur d'objet complet) que transversalement (convertir un pointeur d'objet vers un pointeur d'un autre objet frère dans la hiérarchie de classes).

L'opérateur `dynamic_cast` peut être utilisé dans le but de convertir un pointeur sur une classe de base virtuelle vers une des ses classes filles, ce que ne pouvaient pas faire les transtypages classiques du C. En revanche, il ne peut pas être utilisé afin d'accéder à des classes de base qui ne sont pas visibles (en particulier, les classes de base héritées en `private`).

Exemple 9-2. Opérateur dynamic_cast

```
#include <stdlib.h>

struct A
{
    virtual void f(void)
    {
        return ;
    }
};

struct B : virtual public A
{
};

struct C : virtual public A, public B
{
};

struct D
{
    virtual void g(void)
    {
```

```
        return ;
    }
};

struct E : public B, public C, public D
{
};

int main(void)
{
    E e;        // e contient deux sous-objets de classe B
                // (mais un seul sous-objet de classe A).
                // Les sous-objets de classe C et D sont
                // frères.
    A *pA=&e;   // Dérivation légale : le sous-objet
                // de classe A est unique.
    // C *pC=(C *) pA;// Illégal : A est une classe de base
                      // virtuelle (erreur de compilation).
    C *pC=dynamic_cast<C *>(pA);  // Légal. Transtypage
                                  // dynamique vertical.
    D *pD=dynamic_cast<D *>(pC);  // Légal. Transtypage
                                  // dynamique horizontal.
    B *pB=dynamic_cast<B *>(pA);  // Légal, mais échouera
                                  // à l'exécution (ambiguïté).
    return EXIT_SUCCESS;
}
```

9.2.2. Transtypage statique

Contrairement au transtypage dynamique, le transtypage statique n'effectue aucune vérification des types dynamiques lors du transtypage. Il est donc nettement plus dangereux que le transtypage dynamique. Cependant, contrairement au transtypage C classique, il ne permet toujours pas de supprimer les qualifications de constance.

Le transtypage statique s'effectue à l'aide de l'opérateur `static_cast`, dont la syntaxe est exactement la même que celle de l'opérateur `dynamic_cast` :

```
static_cast<type>(expression)
```

où `type` et `expression` ont la même signification que pour l'opérateur `dynamic_cast`.

Essentiellement, l'opérateur `static_cast` n'effectue l'opération de transtypage que si l'expression suivante est valide :

```
type temporaire(expression);
```

Cette expression construit un objet temporaire quelconque de type type et l'initialise avec la valeur de expression. Contrairement à l'opérateur `dynamic_cast`, l'opérateur `static_cast` permet donc d'effectuer les conversions entre les types autres que les classes définies par l'utilisateur. Aucune vérification de la validité de la conversion n'a lieu cependant (comme pour le transtypage C classique).

Si une telle expression n'est pas valide, le transtypage peut malgré tout avoir lieu s'il s'agit d'un transtypage entre classes dérivées et classes de base. L'opérateur `static_cast` permet d'effectuer les transtypages de ce type dans les deux sens (classe de base vers classe dérivée et classe dérivée

vers classe de base). Le transtypage d'une classe de base vers une classe dérivée ne doit être fait que lorsqu'on est sûr qu'il n'y a pas de danger, puisqu'aucune vérification dynamique n'a lieu avec `static_cast`.

Enfin, toutes les expressions peuvent être converties en void avec des qualifications de constance et de volatilité. Cette opération a simplement pour but de supprimer la valeur de l'expression (puisque void représente le type vide).

9.2.3. Transtypage de constance et de volatilité

La suppression des attributs de constance et de volatilité peut être réalisée grâce à l'opérateur `const_cast`. Cet opérateur suit exactement la même syntaxe que les opérateurs `dynamic_cast` et `static_cast` :

```
const_cast<type>(expression)
```

L'opérateur `const_cast` peut travailler essentiellement avec des références et des pointeurs. Il permet de réaliser les transtypages dont le type destination est moins contraint que le type source vis-à-vis des mots clés `const` et `volatile`.

En revanche, l'opérateur `const_cast` ne permet pas d'effectuer d'autres conversions que les autres opérateurs de transtypage (ou simplement les transtypages C classiques) peuvent réaliser. Par exemple, il est impossible de l'utiliser pour convertir un flottant en entier. Lorsqu'il travaille avec des références, l'opérateur `const_cast` vérifie que le transtypage est légal en convertissant les références en pointeurs et en regardant si le transtypage n'implique que les attributs `const` et `volatile`. `const_cast` ne permet pas de convertir les pointeurs de fonctions.

9.2.4. Réinterprétation des données

L'opérateur de transtypage le plus dangereux est `reinterpret_cast`. Sa syntaxe est la même que celle des autres opérateurs de transtypage `dynamic_cast`, `static_cast` et `const_cast` :

```
reinterpret_cast<type>(expression)
```

Cet opérateur permet de réinterpréter les données d'un type en un autre type. Aucune vérification de la validité de cette opération n'est faite. Ainsi, les lignes suivantes :

```
double f=2.3;
int i=1;
const_cast<int &>(f)=i;
```

sont strictement équivalentes aux lignes suivantes :

```
double f=2.3;
int i=1;
*((int *) &f)=i;
```

L'opérateur `reinterpret_cast` doit cependant respecter les règles suivantes :

- il ne doit pas permettre la suppression des attributs de constance et de volatilité ;
- il doit être symétrique (c'est-à-dire que la réinterprétation d'un type T1 en tant que type T2, puis la réinterprétation du résultat en type T1 doit redonner l'objet initial).

Chapitre 10. Les espaces de nommage

Les *espaces de nommage* sont des zones de déclaration qui permettent de délimiter la recherche des noms des identificateurs par le compilateur. Leur but est essentiellement de regrouper les identificateurs logiquement et d'éviter les conflits de noms entre plusieurs parties d'un même projet. Par exemple, si deux programmeurs définissent différemment une même structure dans deux fichiers différents, un conflit entre ces deux structures aura lieu au mieux à l'édition de liens, et au pire lors de l'utilisation commune des sources de ces deux programmeurs. Ce type de conflit provient du fait que le C++ ne fournit qu'un seul espace de nommage de portée globale, dans lequel il ne doit y avoir aucun conflit de nom. Grâce aux espaces de nommage non globaux, ce type de problème peut être plus facilement évité, parce que l'on peut éviter de définir les objets globaux dans la portée globale.

10.1. Définition des espaces de nommage

10.1.1. Espaces de nommage nommés

Lorsque le programmeur donne un nom à un espace de nommage, celui-ci est appelé un *espace de nommage nommé*. La syntaxe de ce type d'espace de nommage est la suivante :

```
namespace nom
{
    déclarations | définitions
}
```

`nom` est le nom de l'espace de nommage, et `déclarations` et `définitions` sont les déclarations et les définitions des identificateurs qui lui appartiennent.

Contrairement aux régions déclaratives classiques du langage (comme par exemple les classes), un `namespace` peut être découpé en plusieurs morceaux. Le premier morceaux sert de déclaration, et les suivants d'extensions. La syntaxe pour une extension d'espace de nommage est exactement la même que celle de la partie de déclaration.

Exemple 10-1. Extension de namespace

```
namespace A   // Déclaration de l'espace de nommage A.
{
   int i;
}

namespace B   // Déclaration de l'espace de nommage B.
{
    int i;
}

namespace A   // Extension de l'espace de nommage A.
{
    int j;
}
```

Les identificateurs déclarés ou définis à l'intérieur d'un même espace de nommage ne doivent pas entrer en conflit. Ils peuvent avoir les mêmes noms, mais seulement dans le cadre de la surcharge. Un

espace de nommage se comporte donc exactement comme les zones de déclaration des classes et de la portée globale.

L'accès aux identificateurs des espaces de nommage se fait par défaut grâce à l'opérateur de résolution de portée (`::`), et en qualifiant le nom de l'identificateur à utiliser du nom de son espace de nommage. Cependant, cette qualification est inutile à l'intérieur de l'espace de nommage lui-même, exactement comme pour les membres des classes à l'intérieur de leur classe.

Exemple 10-2. Accès aux membres d'un namespace

```
#include <stdlib.h>

int i=1;    // i est global.

namespace A
{
    int i=2; // i de l'espace de nommage A.
    int j=i; // Utilise A::i.
}

int main(void)
{
    i=1;     // Utilise ::i.
    A::i=3;  // Utilise A::i.
    return EXIT_SUCCESS;
}
```

Les fonctions membres d'un espace de nommage peuvent être définies à l'intérieur de cet espace, exactement comme les fonctions membres de classes. Elles peuvent également être définies en dehors de cet espace, si l'on utilise l'opérateur de résolution de portée. Les fonctions ainsi définies doivent apparaître après leur déclaration dans l'espace de nommage.

Exemple 10-3. Définition externe d'une fonction de namespace

```
namespace A
{
    int f(void);   // Déclaration de A::f.
}

int A::f(void)     // Définition de A::f.
{
    return 0;
}
```

Il est possible de définir un espace de nommage à l'intérieur d'un autre espace de nommage. Cependant, cette déclaration doit obligatoirement avoir lieu au niveau déclaratif le plus externe de l'espace de nommage qui contient le sous-espace de nommage. On ne peut donc pas déclarer d'espaces de nommage à l'intérieur d'une fonction ou à l'intérieur d'une classe.

Exemple 10-4. Définition de namespace dans un namespace

```
namespace Conteneur
{
    int i;              // Conteneur::i.
    namespace Contenu
    {
```

```
        int j;          // Conteneur::Contenu::j.
    }
}
```

10.1.2. Espaces de nommage anonymes

Lorsque, lors de la déclaration d'un espace de nommage, aucun nom n'est donné, un *espace de nommage anonyme* est créé. Ce type d'espace de nommage permet d'assurer l'unicité du nom de l'espace de nommage ainsi déclaré. Les espaces de nommage anonymes peuvent donc remplacer efficacement le mot clé `static` pour rendre unique des identificateurs dans un fichier. Cependant, elles sont plus puissantes, parce que l'on peut également déclarer des espaces de nommage anonymes à l'intérieur d'autres espaces de nommage.

Exemple 10-5. Définition de namespace anonyme

```
namespace
{
    int i;          // Équivalent à unique::i;
}
```

Dans l'exemple précédent, la déclaration de `i` se fait dans un espace de nommage dont le nom est choisi par le compilateur de manière unique. Cependant, comme on ne connaît pas ce nom, le compilateur utilise une directive `using` (voir plus loin) afin de pouvoir utiliser les identificateurs de cet espace de nommage anonyme sans préciser leur nom complet avec l'opérateur de résolution de portée.

Si, dans un espace de nommage, un identificateur est déclaré avec le même nom qu'un autre identificateur déclaré dans un espace de nommage plus global, l'identificateur global est masqué. De plus, l'identificateur ainsi défini ne peut être accédé en dehors de son espace de nommage que par un nom complètement qualifié à l'aide de l'opérateur de résolution de portée. Toutefois, si l'espace de nommage dans lequel il est défini est un espace de nommage anonyme, cet identificateur ne pourra pas être référencé, puisqu'on ne peut pas préciser le nom des espaces de nommage anonymes.

Exemple 10-6. Ambiguïtés entre espaces de nommage

```
namespace
{
    int i;           // Déclare unique::i.
}

void f(void)
{
    ++i;             // Utilise unique::i.
}

namespace A
{
    namespace
    {
        int i;      // Définit A::unique::i.
        int j;      // Définit A::unique::j.
    }

    void g(void)
    {
```

```
        ++i;       // Utilise A::unique::i.
        ++A::i;    // Utilise A::unique::i.
        ++j;       // Utiliser A::unique::j.
    }
}
```

Les identificateurs déclarés dans un espace de nommage anonyme ne sont visibles que du fichier courant. Ils ne peuvent en effet être vus des autres unités de compilation, puisque pour cela il faudrait connaître le nom utilisé par le compilateur pour l'espace de nommage anonyme. De ce fait, les espaces de nommage anonymes constituent une technique de remplacement de l'utilisation de la classe de stockage `static` pour les identificateurs globaux.

10.1.3. Alias d'espaces de nommage

Lorsqu'un espace de nommage porte un nom très compliqué, il peut être avantageux de définir un *alias* pour ce nom. L'alias aura alors un nom plus simple.

Cette opération peut être réalisée à l'aide de la syntaxe suivante :

```
namespace nom_alias = nom;
```

`nom_alias` est ici le nom de l'alias de l'espace de nommage, et `nom` est le nom de l'espace de nommage lui-même.

Les noms donnés aux alias d'espaces de nommage ne doivent pas entrer en conflit avec les noms des autres identificateurs du même espace de nommage, que celui-ci soit l'espace de nommage de portée globale ou non.

10.2. Déclaration using

Les *déclarations using* permettent d'utiliser un identificateur d'un espace de nommage de manière simplifiée, sans avoir à spécifier son nom complet (c'est-à-dire le nom de l'espace de nommage suivi du nom de l'identificateur).

10.2.1. Syntaxe des déclarations using

La syntaxe des déclarations using est la suivante :

```
using identificateur;
```

où `identificateur` est le nom complet de l'identificateur à utiliser, avec qualification d'espace de nommage.

Exemple 10-7. Déclaration using

```
namespace A
{
    int i;        // Déclare A::i.
    int j;        // Déclare A::j.
}
```

```
void f(void)
{
    using A::i;   // A::i peut être utilisé sous le nom i.
    i=1;          // Équivalent à A::i=1.
    j=1;          // Erreur ! j n'est pas défini !
    return ;
}
```

Les déclarations `using` permettent en fait de déclarer des alias des identificateurs. Ces alias doivent être considérés exactement comme des déclarations normales. Cela signifie qu'ils ne peuvent être déclarés plusieurs fois que lorsque les déclarations multiples sont autorisées (déclarations de variables ou de fonctions en dehors des classes), et de plus ils appartiennent à l'espace de nommage dans lequel ils sont définis.

Exemple 10-8. Déclarations using multiples

```
#include <stdlib.h>

namespace A
{
    int i;
    void f(void)
    {
    }
}

namespace B
{
    using A::i;  // Déclaration de l'alias B::i, qui représente A::i.
    using A::i;  // Légal : double déclaration de A::i.

    using A::f;  // Déclare void B::f(void),
                 // fonction identique à A::f.
}

int main(void)
{
    B::f();      // Appelle A::f.
    return EXIT_SUCCESS;
}
```

L'alias créé par une déclaration `using` permet de référencer uniquement les identificateurs qui sont visibles au moment où la déclaration `using` est faite. Si l'espace de nommage concerné par la déclaration `using` est étendu après cette dernière, les nouveaux identificateurs de même nom que celui de l'alias ne seront pas pris en compte.

Exemple 10-9. Extension de namespace après une déclaration using

```
namespace A
{
    void f(int);
}

using A::f;            // f est synonyme de A::f(int).

namespace A
```

```
{
    void f(char);       // f est toujours synonyme de A::f(int),
                        // mais pas de A::f(char).
}

void g()
{
    f('a');             // Appelle A::f(int), même si A::f(char)
                        // existe.
}
```

Si plusieurs déclarations locales et `using` déclarent des identificateurs de même nom, ou bien ces identificateurs doivent tous se rapporter au même objet, ou bien ils doivent représenter des fonctions ayant des signatures différentes (les fonctions déclarées sont donc surchargées). Dans le cas contraire, des ambiguïtés peuvent apparaître et le compilateur signale une erreur lors de la déclaration `using`.

Exemple 10-10. Conflit entre déclarations using et identificateurs locaux

```
namespace A
{
    int i;
    void f(int);
}

void g(void)
{
    int i;         // Déclaration locale de i.
    using A::i;    // Erreur : i est déjà déclaré.
    void f(char); // Déclaration locale de f(char).
    using A::f;    // Pas d'erreur, il y a surcharge de f.
    return ;
}
```

> **Note :** Ce comportement diffère de celui des directives `using`. En effet, les directives `using` reportent la détection des erreurs à la première utilisation des identificateurs ambigus.

10.2.2. Utilisation des déclarations using dans les classes

Une déclaration `using` peut être utilisée dans la définition d'une classe. Dans ce cas, elle doit se rapporter à une classe de base de la classe dans laquelle elle est utilisée. De plus, l'identificateur donné à la déclaration `using` doit être accessible dans la classe de base (c'est-à-dire de type `protected` ou `public`).

Exemple 10-11. Déclaration using dans une classe

```
namespace A
{
    float f;
}

class Base
{
```

```
    int i;
public:
    int j;
};

class Derivee : public Base
{
    using A::f;      // Illégal : f n'est pas dans une classe
                     // de base.
    using Base::i;   // Interdit : Derivee n'a pas le droit
                     // d'utiliser Base::i.
public:
    using Base::j;   // Légal.
};
```

Dans l'exemple précédent, seule la troisième déclaration est valide, parce que c'est la seule qui se réfère à un membre accessible de la classe de base. Le membre `j` déclaré sera donc un synonyme de `Base::j` dans la classe Derivee.

En général, les membres des classes de base sont accessibles directement. Quelle est donc l'utilité des déclarations `using` dans les classes ? En fait, elles peuvent être utilisées pour modifier les droits d'accès aux membres des classes de base, pourvu que la classe dérivée puisse y accéder bien entendu. Pour cela, il suffit de placer la déclaration `using` dans une zone de déclaration du type désiré.

Exemple 10-12. Modification des droits d'accès à l'aide d'une directive using

```
class Base
{
public:
    int i;
protected:
    int j;
};

class Derivee : protected Base
{
private:
    using Base::i;  // i ne sera plus visible des classes dérivées de Derivee.
public:
    using Base::j;  // j est maintenant publiquement accessible.
};
```

> **Note :** Certains compilateurs interprètent différemment les paragraphe 7.3.3 et 11.3 de la norme C++, qui concerne l'accessibilité des membres introduits avec une déclaration `using`. Certains considèrent que les déclarations `using` ne permettent que de rétablir l'accessibilité des droits sur les membres des classes de base dont l'héritage a restreint l'accès. D'autres ne permettent que de restreindre l'accessibilité et non pas de les modifier. Pour autant, le comportement est parfaitement défini, il permet de modifier les droits d'accès, pas de les restreindre, ni seulement de les rétablir.

Quand une fonction d'une classe de base est introduite dans une classe dérivée à l'aide d'une déclaration `using`, et qu'une fonction de même nom et de même signature est définie dans la classe dérivée, cette dernière fonction surcharge la fonction de la classe de base. Il n'y a pas d'ambiguïté dans ce cas.

10.3. Directive using

La *directive using* permet d'utiliser, sans spécification d'espace de nommage, non pas un identificateur comme dans le cas de la déclaration `using`, mais tous les identificateurs de cet espace de nommage.

La syntaxe de la directive using est la suivante :

```
using namespace nom;
```

où `nom` est le nom de l'espace de nommage dont les identificateurs doivent être utilisés sans qualification complète.

Exemple 10-13. Directive using

```
namespace A
{
    int i;        // Déclare A::i.
    int j;        // Déclare A::j.
}

void f(void)
{
    using namespace A; // On utilise les identificateurs de A.
    i=1;          // Équivalent à A::i=1.
    j=1;          // Équivalent à A::j=1.
    return ;
}
```

Après une directive `using`, il est toujours possible d'utiliser les noms complets des identificateurs de l'espace de nommage, mais ce n'est plus nécessaire. Les directives `using` sont valides à partir de la ligne où elles sont déclarées jusqu'à la fin du bloc de portée courante. Si un espace de nommage est étendu après une directive `using`, les identificateurs définis dans l'extension de l'espace de nommage peuvent être utilisés exactement comme les identificateurs définis avant la directive `using` (c'est-à-dire sans qualification complète de leurs noms).

Exemple 10-14. Extension de namespace après une directive using

```
namespace A
{
    int i;
}

using namespace A;

namespace A
{
    int j;
}

void f(void)
{
    i=0;    // Initialise A::i.
    j=0;    // Initialise A::j.
    return ;
}
```

Il se peut que lors de l'introduction des identificateurs d'un espace de nommage par une directive `using`, des conflits de noms apparaissent. Dans ce cas, aucune erreur n'est signalée lors de la directive `using`. En revanche, une erreur se produit si un des identificateurs pour lesquels il y a conflit est utilisé.

Exemple 10-15. Conflit entre directive using et identificateurs locaux

```
namespace A
{
    int i;  // Définit A::i.
}

namespace B
{
    int i;  // Définit B::i.
    using namespace A;   // A::i et B::i sont en conflit.
                         // Cependant, aucune erreur n'apparaît.
}

void f(void)
{
    using namespace B;
    i=2;    // Erreur : il y a ambiguïté.
    return ;
}
```

Chapitre 11. Les template

11.1. Généralités

Nous avons vu précédemment comment réaliser des structures de données relativement indépendantes de la classe de leurs données (c'est-à-dire de leur type) avec les classes abstraites. Par ailleurs, il est faisable de faire des fonctions travaillant sur de nombreux types grâce à la surcharge. Je rappelle qu'en C++, tous les types sont en fait des classes.

Cependant, l'emploi des classes abstraites est assez fastidieux et a l'inconvénient d'affaiblir le contrôle des types réalisé par le compilateur. De plus, la surcharge n'est pas généralisable pour tous les types de données. Il serait possible d'utiliser des macros pour faire des fonctions atypiques mais cela serait au détriment de la taille du code.

Le C++ permet de résoudre ces problèmes grâce aux paramètres génériques, que l'on appelle encore *paramètres template*. Un paramètre `template` est soit un *type générique*, soit une *constante* dont le type est assimilable à un type intégral. Comme leur nom l'indique, les paramètres `template` permettent de paramétrer la définition des fonctions et des classes. Les fonctions et les classes ainsi paramétrées sont appelées respectivement *fonctions template* et *classes template*.

Les *fonctions template* sont donc des fonctions qui peuvent travailler sur des objets dont le type est un type générique (c'est-à-dire un type quelconque), ou qui peuvent êtres paramétrés par une constante de type intégral. Les *classes template* sont des classes qui contiennent des membres dont le type est générique ou qui dépendent d'un paramètre intégral.

En général, la génération du code a lieu lors d'une opération au cours de laquelle les types génériques sont remplacés par des vrais types et les paramètres de type intégral prennent leur valeur. Cette opération s'appelle l'*instanciation des template*. Elle a lieu lorsqu'on utilise la fonction ou la classe `template` pour la première fois. Les types réels à utiliser à la place des types génériques sont déterminés lors de cette première utilisation par le compilateur, soit implicitement à partir du contexte d'utilisation du `template`, soit par les paramètres donnés explicitement par le programmeur.

11.2. Déclaration des paramètres template

Les paramètres `template` sont, comme on l'a vu, soit des types génériques, soit des constantes dont le type peut être assimilé à un type intégral.

11.2.1. Déclaration des types template

Les `template` qui sont des types génériques sont déclarés par la syntaxe suivante :

```
template <class|typename nom[=type]
         [, class|typename nom[=type]
         [...]>
```

où `nom` est le nom que l'on donne au type générique dans cette déclaration. Le mot clé `class` a ici exactement la signification de « type ». Il peut d'ailleurs être remplacé indifféremment dans cette syntaxe par le mot clé `typename`. La même déclaration peut être utilisée pour déclarer un nombre arbitraire de types génériques, en les séparant par des virgules. Les paramètres `template` qui sont des types peuvent prendre des valeurs par défaut, en faisant suivre le nom du paramètre d'un signe égal et de la valeur. Ici, la valeur par défaut doit évidemment être un type déjà déclaré.

Exemple 11-1. Déclaration de paramètres template

```
template <class T, typename U, class V=int>
```

Dans cet exemple, T, U et V sont des types génériques. Ils peuvent remplacer n'importe quel type du langage déjà déclaré au moment où la déclaration `template` est faite. De plus, le type générique V a pour valeur par défaut le type entier int. On voit bien dans cet exemple que les mots clés `typename` et `class` peuvent être utilisés indifféremment.

Lorsqu'on donne des valeurs par défaut à un type générique, on doit donner des valeurs par défaut à tous les types génériques qui le suivent dans la déclaration `template`. La ligne suivante provoquera donc une erreur de compilation :

```
template <class T=int, class V>
```

Il est possible d'utiliser une classe `template` en tant que type générique. Dans ce cas, la classe doit être déclarée comme étant `template` à l'intérieur même de la déclaration `template`. La syntaxe est donc la suivante :

```
template <template <class Type> class Classe [,...]>
```

où Type est le type générique utilisé dans la déclaration de la classe `template` Classe. On appelle les paramètres `template` qui sont des classes `template` des paramètres *template template*. Rien n'interdit de donner une valeur par défaut à un paramètre `template template` : le type utilisé doit alors être une classe `template` déclarée avant la déclaration `template`.

Exemple 11-2. Déclaration de paramètre template template

```
template <class T>
class Tableau
{
    // Définition de la classe template Tableau.
};

template <class U, class V, template <class T> class C=Tableau>
class Dictionnaire
{
    C<U> Clef;
    C<V> Valeur;
    // Reste de la définition de la classe Dictionnaire.
};
```

Dans cet exemple, la classe `template` Dictionnaire permet de relier des clés à leurs éléments. Ces clés et ces valeurs peuvent prendre n'importe quel type. Les clés et les valeurs sont stockées parallèlement dans les membres *`Clef`* et *`Valeur`*. Ces membres sont en fait des conteneurs `template`, dont la classe est générique et désignée par le paramètre `template template` C. Le paramètre `template` de C est utilisé pour donner le type des données stockées, à savoir les types génériques U et V dans le cas de la classe Dictionnaire. Enfin, la classe Dictionnaire peut utiliser un conteneur par défaut, qui est la classe `template` Tableau.

Pour plus de détails sur la déclaration des classes `template`, voir la Section 11.3.2.

11.2.2. Déclaration des constantes template

La déclaration des paramètres `template` de type constante se fait de la manière suivante :

```
template <type paramètre[=valeur][, ...]>
```

où `type` est le type du paramètre constant, `paramètre` est le nom du paramètre et `valeur` est sa valeur par défaut. Il est possible de donner des paramètres `template` qui sont des types génériques et des paramètres `template` qui sont des constantes dans la même déclaration.

Le type des constantes `template` doit obligatoirement être l'un des types suivants :

- type intégral (char, wchar_t, int, long, short et leurs versions signées et non signées) ou énuméré ;
- pointeur ou référence d'objet ;
- pointeur ou référence de fonction ;
- pointeur sur membre.

Ce sont donc tous les types qui peuvent être assimilés à des valeurs entières (entiers, énumérés ou adresses).

Exemple 11-3. Déclaration de paramètres template de type constante

```
template <class T, int i, void (*f)(int)>
```

Cette déclaration `template` comprend un type générique T, une constante `template i` de type int, et une constante `template f` de type pointeur sur fonction prenant un entier en paramètre et ne renvoyant rien.

> **Note :** Les paramètres constants de type référence ne peuvent pas être initialisés avec une donnée immédiate ou une donnée temporaire lors de l'instanciation du `template`. Voir la Section 11.4 pour plus de détails sur l'instanciation des `template`.

11.3. Fonctions et classes template

Après la déclaration d'un ou de plusieurs paramètres `template` suit en général la déclaration ou la définition d'une fonction ou d'une classe `template`. Dans cette définition, les types génériques peuvent être utilisés exactement comme s'il s'agissait de types normaux. Les constantes `template` peuvent être utilisées dans la fonction ou la classe `template` comme des constantes locales.

11.3.1. Fonctions template

La déclaration et la définition des fonctions `template` se fait exactement comme si la fonction était une fonction normale, à ceci près qu'elle doit être précédée de la déclaration des paramètres `template`. La syntaxe d'une déclaration de fonction `template` est donc la suivante :

```
template <paramètres_template>
type fonction(paramètres_fonction);
```

où `paramètre_template` est la liste des paramètres `template` et `paramètres_fonction` est la liste des paramètres de la fonction `fonction`. `type` est le type de la valeur de retour de la fonction, ce peut être un des types génériques de la liste des paramètres `template`.

Tous les paramètres `template` qui sont des types doivent être utilisés dans la liste des paramètres de la fonction, à moins qu'une instanciation explicite de la fonction ne soit utilisée. Cela permet au compilateur de réaliser l'identification des types génériques avec les types à utiliser lors de l'instanciation de la fonction. Voir la Section 11.4 pour plus de détails à ce sujet.

La définition d'une fonction `template` se fait comme une déclaration avec le corps de la fonction. Il est alors possible d'y utiliser les paramètres `template` comme s'ils étaient des types normaux : des variables peuvent être déclarées avec un type générique, et les constantes `template` peuvent être utilisées comme des variables définies localement avec la classe de stockage const. Les fonctions `template` s'écrivent donc exactement comme des fonctions classiques.

Exemple 11-4. Définition de fonction template

```
template <class T>
T Min(T x, T y)
{
    return x<y ? x : y;
}
```

La fonction `Min` ainsi définie fonctionnera parfaitement pour toute classe pour laquelle l'opérateur < est défini. Le compilateur déterminera automatiquement quel est l'opérateur à employer pour chaque fonction `Min` qu'il rencontrera.

Les fonctions `template` peuvent être surchargées, aussi bien par des fonctions classiques que par d'autres fonctions `template`. Lorsqu'il y a ambiguïté entre une fonction `template` et une fonction normale qui la surcharge, toutes les références sur le nom commun à ces fonctions se rapporteront à la fonction classique.

Une fonction `template` peut être déclarée amie d'une classe, `template` ou non, pourvu que cette classe ne soit pas locale. Toutes les instances générées à partir d'une fonction amie `template` sont amies de la classe donnant l'amitié, et ont donc libre accès sur toutes les données de cette classe.

11.3.2. Les classes template

La déclaration et la définition d'une classe `template` se font comme celles d'une fonction `template` : elles doivent être précédées de la déclaration `template` des types génériques. La déclaration suit donc la syntaxe suivante :

```
template <paramètres_template>
class|struct|union nom;
```

où `paramètres_template` est la liste des paramètres `template` utilisés par la classe `template` `nom`.

La seule particularité dans la définition des classes `template` est que si les méthodes de la classe ne sont pas définies dans la déclaration de la classe, elles devront elles aussi être déclarées `template` :

```
template <paramètres_template>
type classe<paramètres>::nom(paramètres_méthode)
{
    ...
}
```

où `paramètre_template` représente la liste des paramètres `template` de la classe `template classe`, `nom` représente le nom de la méthode à définir, et `paramètres_méthode` ses paramètres.

Il est absolument nécessaire dans ce cas de spécifier tous les paramètres `template` de la liste `paramètres_template` dans paramètres, séparés par des virgules, afin de caractériser le fait que c'est la classe `classe` qui est `template` et qu'il ne s'agit pas d'une méthode `template` d'une classe normale. D'une manière générale, il faudra toujours spécifier les types génériques de la classe entre les signes d'infériorité et de supériorité, juste après son nom, à chaque fois qu'on voudra la référencer. Cette règle est cependant facultative lorsque la classe est référencée à l'intérieur d'une fonction membre.

Contrairement aux fonctions `template` non membres, les méthodes des classes `template` peuvent utiliser des types génériques de leur classe sans pour autant qu'ils soient utilisés dans la liste de leurs paramètres. En effet, le compilateur détermine quels sont les types à identifier aux types génériques lors de l'instanciation de la classe `template`, et n'a donc pas besoin d'effectuer cette identification avec les types des paramètres utilisés. Voir la Section 11.3.3 pour plus de détails à ce sujet.

Exemple 11-5. Définition d'une pile template

```
template <class T>
class Stack
{
    typedef struct stackitem
    {
        T Item;                 // On utilise le type T comme
        struct stackitem *Next; // si c'était un type normal.
    } StackItem;

    StackItem *Tete;

public:         // Les fonctions de la pile :
    Stack(void);
    Stack(const Stack<T> &);
                // La classe est référencée en indiquant
                // son type entre < et > ("Stack<T>").
                // Ici, ce n'est pas une nécessité
                // cependant.
    ~Stack(void);
    Stack<T> &operator=(const Stack<T> &);
    void push(T);
    T pop(void);
    bool is_empty(void) const;
    void flush(void);
};

// Pour les fonctions membres définies en dehors de la déclaration
// de la classe, il faut une déclaration de type générique :

template <class T>
Stack<T>::Stack(void) // La classe est référencée en indiquant
                      // son type entre < et > ("Stack<T>").
                      // C'est impératif en dehors de la
                      // déclaration de la classe.
{
    Tete = NULL;
    return;
}
```

```
template <class T>
Stack<T>::Stack(const Stack<T> &Init)
{
    Tete = NULL;
    StackItem *tmp1 = Init.Tete, *tmp2 = NULL;
    while (tmp1!=NULL)
    {
        if (tmp2==NULL)
        {
            Tete= new StackItem;
            tmp2 = Tete;
        }
        else
        {
            tmp2->Next = new StackItem;
            tmp2 = tmp2->Next;
        }
        tmp2->Item = tmp1->Item;
        tmp1 = tmp1->Next;
    }
    if (tmp2!=NULL) tmp2->Next = NULL;
    return;
}

template <class T>
Stack<T>::~Stack(void)
{
    flush();
    return;
}

template <class T>
Stack<T> &Stack<T>::operator=(const Stack<T> &Init)
{
    flush();
    StackItem *tmp1 = Init.Tete, *tmp2 = NULL;

    while (tmp1!=NULL)
    {
        if (tmp2==NULL)
        {
            Tete = new StackItem;
            tmp2 = Tete;
        }
        else
        {
            tmp2->Next = new StackItem;
            tmp2 = tmp2->Next;
        }
        tmp2->Item = tmp1->Item;
        tmp1 = tmp1->Next;
    }
    if (tmp2!=NULL) tmp2->Next = NULL;
    return *this;
}
```

```
template <class T>
void Stack<T>::push(T Item)
{
    StackItem *tmp = new StackItem;
    tmp->Item = Item;
    tmp->Next = Tete;
    Tete = tmp;
    return;
}

template <class T>
T Stack<T>::pop(void)
{
    T tmp;
    StackItem *ptmp = Tete;

    if (Tete!=NULL)
    {
        tmp = Tete->Item;
        Tete = Tete->Next;
        delete ptmp;
    }
    return tmp;
}

template <class T>
bool Stack<T>::is_empty(void) const
{
    return (Tete==NULL);
}

template <class T>
void Stack<T>::flush(void)
{
    while (Tete!=NULL) pop();
    return;
}
```

Les classes `template` peuvent parfaitement avoir des fonctions amies, que ces fonctions soient elles-mêmes `template` ou non.

11.3.3. Fonctions membres template

Les destructeurs mis à part, les méthodes d'une classe peuvent être `template`, que la classe elle-même soit `template` ou non, pourvu que la classe ne soit pas une classe locale.

Les fonctions membres `template` peuvent appartenir à une classe `template` ou à une classe normale.

Lorsque la classe à laquelle elles appartiennent n'est pas `template`, leur syntaxe est exactement la même que pour les fonctions `template` non membre.

Exemple 11-6. Fonction membre template

```
class A
{
    int i;   // Valeur de la classe.
public:
    template <class T>
    void add(T valeur);
};

template <class T>
void A::add(T valeur)
{
    i=i+((int) valeur);   // Ajoute valeur à A::i.
    return ;
}
```

Si, en revanche, la classe dont la fonction membre fait partie est elle aussi `template`, il faut spécifier deux fois la syntaxe `template` : une fois pour la classe, et une fois pour la fonction. Si la fonction membre `template` est définie à l'intérieur de la classe, il n'est pas nécessaire de donner les paramètres `template` de la classe, et la définition de la fonction membre `template` se fait donc exactement comme celle d'une fonction `template` classique.

Exemple 11-7. Fonction membre template d'une classe template

```
template<class T>
class Chaine
{
public:
    // Fonction membre template définie
    // à l'extérieur de la classe template :

    template<class T2> int compare(const T2 &);

    // Fonction membre template définie
    // à l'intérieur de la classe template :

    template<class T2>
    Chaine(const Chaine<T2> &s)
    {
        // ...
    }
};

// À l'extérieur de la classe template, on doit donner
// les déclarations template pour la classe
// et pour la fonction membre template :

template<class T> template<class T2>
int Chaine<T>::compare(const T2 &s)
{
    // ...
}
```

Les fonctions membres virtuelles ne peuvent pas être `template`. Si une fonction membre `template` a le même nom qu'une fonction membre virtuelle d'une classe de base, elle ne constitue pas une redéfinition de cette fonction. Par conséquent, les mécanismes de virtualité sont inutilisables avec les

fonctions membres `template`. On peut contourner ce problème de la manière suivante : on définira une fonction membre virtuelle non `template` qui appellera la fonction membre `template`.

Exemple 11-8. Fonction membre template et fonction membre virtuelle

```
class B
{
    virtual void f(int);
};

class D : public B
{
    template <class T>
    void f(T);      // Cette fonction ne redéfinit pas B::f(int).

    void f(int i)  // Cette fonction surcharge B::f(int).
    {
        f<>(i);    // Elle appelle de la fonction template.
        return ;
    }
};
```

Dans l'exemple précédent, on est obligé de préciser que la fonction à appeler dans la fonction virtuelle est la fonction `template`, et qu'il ne s'agit donc pas d'un appel récursif de la fonction virtuelle. Pour cela, on fait suivre le nom de la fonction `template` d'une paire de signes inférieur et supérieur.

Plus généralement, si une fonction membre `template` d'une classe peut être spécialisée en une fonction qui a la même signature qu'une autre fonction membre de la même classe, et que ces deux fonctions ont le même nom, toute référence à ce nom utilisera la fonction non-`template`. Il est possible de passer outre cette règle, à condition de donner explicitement la liste des paramètres `template` entre les signes inférieur et supérieur lors de l'appel de la fonction.

Exemple 11-9. Surcharge de fonction membre par une fonction membre template

```
#include <stdlib.h>
#include <iostream>
using namespace std;

struct A
{
    void f(int);

    template <class T>
    void f(T)
    {
        cout << "Template" << endl;
    }
};

// Fonction non template :
void A::f(int)
{
    cout << "Non template" << endl;
}

// Fonction template :
```

```
template <>
void A::f<int>(int)
{
    cout << "Spécialisation f<int>" << endl;
}

int main(void)
{
    A a;
    a.f(1);    // Appel de la version non-template de f.
    a.f('c');  // Appel de la version template de f.
    a.f<>(1); // Appel de la version template spécialisée de f.
    return EXIT_SUCCESS;
}
```

Pour plus de détails sur la spécialisation des `template`, voir la Section 11.5.

11.4. Instanciation des template

La définition des fonctions et des classes `template` ne génère aucun code tant que tous les paramètres `template` n'ont pas pris chacun une valeur spécifique. Il faut donc, lors de l'utilisation d'une fonction ou d'une classe `template`, fournir les valeurs pour tous les paramètres qui n'ont pas de valeur par défaut. Lorsque suffisamment de valeurs sont données, le code est généré pour ce jeu de valeurs. On appelle cette opération l'*instanciation des template*.

Plusieurs possibilités sont offertes pour parvenir à ce résultat : l'*instanciation implicite* et l'*instanciation explicite*.

11.4.1. Instanciation implicite

L'*instanciation implicite* est utilisée par le compilateur lorsqu'il rencontre une expression qui utilise pour la première fois une fonction ou une classe `template`, et qu'il doit l'instancier pour continuer son travail. Le compilateur se base alors sur le contexte courant pour déterminer les types des paramètres `template` à utiliser. Si aucune ambiguïté n'a lieu, il génère le code pour ce jeu de paramètres.

La détermination des types des paramètres `template` peut se faire simplement, ou être déduite de l'expression à compiler. Par exemple, les fonctions membres `template` sont instanciées en fonction du type de leurs paramètres. Si l'on reprend l'exemple de la fonction `template Min` définie dans l'Exemple 11-4, c'est son utilisation directe qui provoque une instanciation implicite.

Exemple 11-10. Instanciation implicite de fonction template

```
int i=Min(2,3);
```

Dans cet exemple, la fonction `Min` est appelée avec les paramètres `2` et `3`. Comme ces entiers sont tous les deux de type int, la fonction `template Min` est instanciée pour le type int. Partout dans la définition de `Min`, le type générique T est donc remplacé par le type int.

Si l'on appelle une fonction `template` avec un jeu de paramètres qui provoque une ambiguïté, le compilateur signale une erreur. Cette erreur peut être levée en surchargeant la fonction `template` par une fonction qui accepte les mêmes paramètres. Par example, la fonction `template Min` ne peut pas être instanciée dans le code suivant :

```
int i=Min(2,3.0);
```

parce que le compilateur ne peut pas déterminer si le type générique T doit prendre la valeur int ou double. Il y a donc une erreur, sauf si une fonction `Min(int, double)` est définie quelque part. Pour résoudre ce type de problème, on devra spécifier manuellement les paramètres `template` de la fonction, lors de l'appel. Ainsi, la ligne précédente compile si on la réécrit comme suit :

```
int i=Min<int>(2,3.0);
```

dans cet exemple, le paramètre `template` est forcé à int, et `3.0` est converti en entier.

On prendra garde au fait que le compilateur utilise une politique minimaliste pour l'instanciation implicite des `template`. Cela signifie qu'il ne créera que le code nécessaire pour compiler l'expression qui exige une instanciation implicite. Par exemple, la définition d'un objet d'une classe `template` dont tous les types définis provoque l'instanciation de cette classe, mais la définition d'un pointeur sur cette classe ne le fait pas. L'instanciation aura lieu lorsqu'un déréférencement sera fait par l'intermédiaire de ce pointeur. De même, seules les fonctionnalités utilisées de la classe `template` seront effectivement définies dans le programme final.

Par exemple, dans le programme suivant :

```
#include <stdlib.h>
#include <iostream>
using namespace std;

template <class T>
class A
{
public:
     void f(void);
     void g(void);
};

// Définition de la méthode A<T>::f() :
template <class T>
void A<T>::f(void)
{
     cout << "A<T>::f() appelée" << endl;
}

// On ne définit pas la méthode A<T>::g()...

int main(void)
{
     A<char> a;  // Instanciation de A<char>.
     a.f();       // Instanciation de A<char>::f().
     return EXIT_SUCCESS;
}
```

seule la méthode `f` de la classe `template` A est instanciée, car c'est la seule méthode utilisée à cet endroit. Ce programme pourra donc parfaitement être compilé, même si la méthode `g` n'a pas été définie.

11.4.2. Instanciation explicite

L'instanciation explicite des `template` est une technique permettant au programmeur de forcer l'instanciation des `template` dans son programme. Pour réaliser une instanciation explicite, il faut spécifier explicitement tous les paramètres `template` à utiliser. Cela se fait simplement en donnant la déclaration du `template`, précédée par le mot clé `template` :

```
template nom<valeur[, valeur[...]]>;
```

Par exemple, pour forcer l'instanciation d'une pile telle que celle définie dans l'Exemple 11-5, il faudra préciser le type des éléments entre crochets après le nom de la classe :

```
template Stack<int>;  // Instancie la classe Stack<int>.
```

Cette syntaxe peut être simplifiée pour les fonctions `template`, à condition que tous les paramètres `template` puissent être déduits par le compilateur des types des paramètres utilisés dans la déclaration de la fonction. Ainsi, il est possible de forcer l'instanciation de la fonction `template Min` de la manière suivante :

```
template int Min(int, int);
```

Dans cet exemple, la fonction `template Min` est instanciée pour le type int, puisque ses paramètres sont de ce type.

Lorsqu'une fonction ou une classe `template` a des valeurs par défaut pour ses paramètres `template`, il n'est pas nécessaire de donner une valeur pour ces paramètres. Si toutes les valeurs par défaut sont utilisées, la liste des valeurs peut être vide (mais les signes d'infériorité et de supériorité doivent malgré tout être présents).

Exemple 11-11. Instanciation explicite de classe template

```
template<class T = char>
class Chaine;

template Chaine<>;  // Instanciation explicite de Chaine<char>.
```

11.4.3. Problèmes soulevés par l'instanciation des template

Les `template` doivent impérativement être définis lors de leur instanciation pour que le compilateur puisse générer le code de l'instance. Cela signifie que les fichiers d'en-tête doivent contenir non seulement la déclaration, mais également la définition complète des `template`. Cela a plusieurs inconvénients. Le premier est bien entendu que l'on ne peut pas considérer les `template` comme les fonctions et les classes normales du langage, pour lesquels il est possible de séparer la déclaration de la définition dans des fichiers séparés. Le deuxième inconvénient est que les instances des `template` sont compilées plusieurs fois, ce qui diminue d'autant plus les performances des compilateurs. Enfin, ce qui est le plus grave, c'est que les instances des `template` sont en multiples exemplaires dans les fichiers objets générés par le compilateur, et accroissent donc la taille des fichiers exécutables à l'issue

de l'édition de liens. Cela n'est pas gênant pour les petits programmes, mais peut devenir rédhibitoire pour les programmes assez gros.

Le premier problème n'est pas trop gênant, car il réduit le nombre de fichiers sources, ce qui n'est en général pas une mauvaise chose. Notez également que les `template` ne peuvent pas être considérés comme des fichiers sources classiques, puisque sans instanciation, ils ne génèrent aucun code machine (ce sont des *classes de classes*, ou « *métaclasses* »). Mais ce problème peut devenir ennuyant dans le cas de bibliothèques `template` écrites et vendues par des sociétés désireuses de conserver leur savoir-faire. Pour résoudre ce problème, le langage donne la possibilité d'exporter les définitions des `template` dans des fichiers complémentaires. Nous verrons la manière de procéder dans la Section 11.7.

Le deuxième problème peut être résolu avec l'exportation des `template`, ou par tout autre technique d'optimisation des compilateurs. Actuellement, la plupart des compilateurs sont capables de générer des fichiers d'en-tête *précompilés*, qui contiennent le résultat de l'analyse des fichiers d'en-tête déjà lus. Cette technique permet de diminuer considérablement les temps de compilation, mais nécessite souvent d'utiliser toujours le même fichier d'en-tête au début des fichiers sources.

Le troisième problème est en général résolu par des techniques variées, qui nécessitent des traitements complexes dans l'éditeur de liens ou le compilateur. La technique la plus simple, utilisée par la plupart des compilateurs actuels, passe par une modification de l'éditeur de liens pour qu'il regroupe les différentes instances des mêmes `template`. D'autres compilateurs, plus rares, gèrent une base de données dans laquelle les instances de `template` générées lors de la compilation sont stockées. Lors de l'édition de liens, les instances de cette base sont ajoutées à la ligne de commande de l'éditeur de liens afin de résoudre les symboles non définis. Enfin, certains compilateurs permettent de désactiver les instanciations implicites des `template`. Cela permet de laisser au programmeur la responsabilité de les instancier manuellement, à l'aide d'instanciations explicites. Ainsi, les `template` peuvent n'être définies que dans un seul fichier source, réservé à cet effet. Cette dernière solution est de loin la plus sûre, et il est donc recommandé d'écrire un tel fichier pour chaque programme.

Ce paragraphe vous a présenté trois des principaux problèmes soulevés par l'utilisation des `template`, ainsi que les solutions les plus courantes qui y ont été apportées. Il est vivement recommandé de consulter la documentation fournie avec l'environnement de développement utilisé, afin à la fois de réduire les temps de compilation et d'optimiser les exécutables générés.

11.5. Spécialisation des template

Jusqu'à présent, nous avons défini les classes et les fonctions `template` d'une manière unique, pour tous les types et toutes les valeurs des paramètres `template`. Cependant, il peut être intéressant de définir une version particulière d'une classe ou d'une fonction pour un jeu particulier de paramètres `template`.

Par exemple, la pile de l'Exemple 11-5 peut être implémentée beaucoup plus efficacement si elle stocke des pointeurs plutôt que des objets, sauf si les objets sont petits (ou appartiennent à un des types prédéfinis du langage). Il peut être intéressant de manipuler les pointeurs de manière transparente au niveau de la pile, pour que la méthode `pop` renvoie toujours un objet, que la pile stocke des pointeurs ou des objets. Afin de réaliser cela, il faut donner une deuxième version de la pile pour les pointeurs.

Le C++ permet tout cela : lorsqu'une fonction ou une classe `template` a été définie, il est possible de la *spécialiser* pour un certain jeu de paramètres `template`. Il existe deux types de spécialisation : les *spécialisations totales*, qui sont les spécialisations pour lesquelles il n'y a plus aucun paramètre `template` (ils ont tous une valeur bien déterminée), et les *spécialisations partielles*, pour lesquelles seuls quelques paramètres `template` ont une valeur fixée.

11.5.1. Spécialisation totale

Les *spécialisations totales* nécessitent de fournir les valeurs des paramètres `template`, séparées par des virgules et entre les signes d'infériorité et de supériorité, après le nom de la fonction ou de la classe `template`. Il faut faire précéder la définition de cette fonction ou de cette classe par la ligne suivante :

```
template <>
```

qui permet de signaler que la liste des paramètres `template` pour cette spécialisation est vide (et donc que la spécialisation est totale).

Par exemple, si la fonction `Min` définie dans l'Exemple 11-4 doit être utilisée sur une structure Structure et se baser sur un des champs de cette structure pour effectuer les comparaisons, elle pourra être spécialisée de la manière suivante :

Exemple 11-12. Spécialisation totale

```
struct Structure
{
    int Clef;     // Clef permettant de retrouver des données.
    void *pData;  // Pointeur sur les données.
};

template <>
Structure Min<Structure>(Structure s1, Structure s2)
{
    if (s1.Clef>s2.Clef)
        return s1;
    else
        return s2;
}
```

> **Note :** Pour quelques compilateurs, la ligne déclarant la liste vide des paramètres `template` ne doit pas être écrite. On doit donc faire des spécialisations totale sans le mot clé `template`. Ce comportement n'est pas celui spécifié par la norme, et le code écrit pour ces compilateurs n'est donc pas portable.

11.5.2. Spécialisation partielle

Les *spécialisations partielles* permettent de définir l'implémentation d'une fonction ou d'une classe `template` pour certaines valeurs de leurs paramètres `template` et de garder d'autres paramètres indéfinis. Il est même possible de changer la nature d'un paramètre `template` (c'est-à-dire préciser s'il s'agit d'un pointeur ou non) et de forcer le compilateur à prendre une implémentation plutôt qu'une autre selon que la valeur utilisée pour ce paramètre est elle-même un pointeur ou non.

Comme pour les spécialisations totales, il est nécessaire de déclarer la liste des paramètres `template` utilisés par la spécialisation. Cependant, à la différence des spécialisations totales, cette liste ne peut plus être vide.

Comme pour les spécialisations totales, la définition de la classe ou de la fonction `template` doit utiliser les signes d'infériorité et de supériorité pour donner la liste des valeurs des paramètres `template` pour la spécialisation.

Exemple 11-13. Spécialisation partielle

```
// Définition d'une classe template :
template <class T1, class T2, int I>
class A
{
};

// Spécialisation n°1 de la classe :
template <class T, int I>
class A<T, T*, I>
{
};

// Spécialisation n°2 de la classe :
template <class T1, class T2, int I>
class A<T1*, T2, I>
{
};

// Spécialisation n°3 de la classe :
template <class T>
class A<int, T*, 5>
{
};

// Spécialisation n°4 de la classe :
template <class T1, class T2, int I>
class A<T1, T2*, I>
{
};
```

On notera que le nombre des paramètres `template` déclarés à la suite du mot clé `template` peut varier, mais que le nombre de valeurs fournies pour la spécialisation est toujours constant (dans l'exemple précédent, il y en a trois).

Les valeurs utilisées dans les identificateurs `template` des spécialisations doivent respecter les règles suivantes :

- une valeur ne peut pas être exprimée en fonction d'un paramètre `template` de la spécialisation ;

```
template <int I, int J>
struct B
{
};

template <int I>
struct B<I, I*2>   // Erreur !
{                   // Spécialisation incorrecte !
};
```

- le type d'une des valeurs de la spécialisation ne peut pas dépendre d'un autre paramètre ;

```
template <class T, T t>
struct C
{
};
```

```
template <class T>
struct C<T, 1>;    // Erreur !
                    // Spécialisation incorrecte !
```

- la liste des arguments de la spécialisation ne doit pas être identique à la liste implicite de la déclaration `template` correspondante.

Enfin, la liste des paramètres `template` de la déclaration d'une spécialisation ne doit pas contenir des valeurs par défaut. On ne pourrait d'ailleurs les utiliser en aucune manière.

11.5.3. Spécialisation d'une méthode d'une classe template

La spécialisation partielle d'une classe peut parfois être assez lourde à employer, en particulier si la structure de données qu'elle contient ne change pas entre les versions spécialisées. Dans ce cas, il peut être plus simple de ne spécialiser que certaines méthodes de la classe et non la classe complète. Cela permet de conserver la définition des méthodes qui n'ont pas lieu d'être modifiées pour les différents types, et d'éviter d'avoir à redéfinir les données membres de la classe à l'identique.

La syntaxe permettant de spécialiser une méthode d'une classe `template` est très simple. Il suffit en effet de considérer la méthode comme une fonction `template` normale, et de la spécialiser en précisant les paramètres `template` à utiliser pour cette spécialisation.

Exemple 11-14. Spécialisation de fonction membre de classe template

```
#include <iostream>

using namespace std;

template <class T>
class Item
{
    T item;
public:
    Item(T);
    void set(T);
    T get(void) const;
    void print(void) const;
};

template <class T>
Item<T>::Item(T i)               // Constructeur
{
    item = i;
}

// Accesseurs :

template <class T>
void Item<T>::set(T i)
{
    item = i;
}
```

```
template <class T>
T Item<T>::get(void) const
{
    return item;
}

// Fonction d'affichage générique :

template <class T>
void Item<T>::print(void) const
{
    cout << item << endl;
}

// Fonction d'affichage spécialisée explicitement pour le type int *
// et la méthode print :
template <>
void Item<int *>::print(void) const
{
    cout << *item << endl;
}
```

11.6. Mot-clé typename

Nous avons déjà vu que le mot clé `typename` pouvait être utilisé pour introduire les types génériques dans les déclarations `template`. Cependant, il peut être utilisé dans un autre contexte pour introduire les identificateurs de types inconnus dans les `template`. En effet, un type générique peut très bien être une classe définie par l'utilisateur, à l'intérieur de laquelle des types sont définis. Afin de pouvoir utiliser ces types dans les définitions des `template`, il est nécessaire d'utiliser le mot clé `typename` pour les introduire, car a priori le compilateur ne sait pas que le type générique contient la définition d'un autre type. Ce mot clé doit être placé avant le nom complet du type :

```
typename identificateur
```

Le mot clé `typename` est donc utilisé pour signaler au compilateur que l'identificateur `identificateur` est un type.

Exemple 11-15. Mot-clé typename

```
class A
{
public:
    typedef int Y;    // Y est un type défini dans la classe A.
};

template <class T>
class X
{
    typename T::Y i;  // La classe template X suppose que le
                      // type générique T définisse un type Y.
};
```

```
X<A> x;              // A peut servir à instancier une classe
                      // à partir de la classe template X.
```

11.7. Fonctions exportées

Comme on l'a vu, les fonctions et classes `template` sont toutes instanciées lorsqu'elles sont rencontrées pour la première fois par le compilateur ou lorsque la liste de leurs paramètres est fournie explicitement.

Cette règle a une conséquence majeure : la définition complète des fonctions et des classes `template` doit être incluse dans chacun des fichiers dans lequel elles sont utilisées. En général, les déclarations et les définitions des fonctions et des classes `template` sont donc regroupées ensemble dans les fichiers d'en-tête (et le code ne se trouve pas dans un fichier C++). Cela est à la fois très lent (la définition doit être relue par le compilateur à chaque fois qu'un `template` est utilisé) et ne permet pas de protéger le savoir faire des entreprises qui éditent des bibliothèques `template`, puisque leur code est accessible à tout le monde.

Afin de résoudre ces problèmes, le C++ permet de « compiler » les fonctions et les classes `template`, et ainsi d'éviter l'inclusion systématique de leur définition dans les fichiers sources. Cette « compilation » se fait à l'aide du mot clé `export`.

Pour parvenir à ce résultat, vous devez déclarer « `export` » les fonctions et les classes `template` concernées. La déclaration d'une classe `template export` revient à déclarer `export` toutes ses fonctions membres non `inline`, toutes ses données statiques, toutes ses classes membres et toutes ses fonctions membres `template` non statiques. Si une fonction `template` est déclarée comme étant `inline`, elle ne peut pas être de type `export`.

Les fonctions et les classes `template` qui sont définies dans un espace de nommage anonyme ne peuvent pas être déclarées `export`. Voir le Chapitre 10 plus de détails sur les espaces de nommage.

Exemple 11-16. Mot-clé export

```
export template <class T>
void f(T);          // Fonction dont le code n'est pas fourni
                    // dans les fichiers qui l'utilisent.
```

Dans cet exemple, la fonction `f` est déclarée `export`. Sa définition est fournie dans un autre fichier, et n'a pas besoin d'être fournie pour que `f` soit utilisable.

Les définitions des fonctions et des classes déclarées `export` doivent elles aussi utiliser le mot clé `export`. Ainsi, la définition de `f` pourra ressembler aux lignes suivantes :

```
export template <class T>
void f(T p)
{
    // Corps de la fonction.
    return ;
}
```

Note : Aucun compilateur ne gère le mot clé `export` à ce jour.

Chapitre 12. Conventions de codage et techniques de base

L'apprentissage de la syntaxe d'un langage est certainement la chose la plus facile à faire en programmation. Mais connaître la syntaxe est loin d'être suffisant pour réaliser des programmes performants, fiables et élégants. La programmation nécessite en effet de bien connaître les techniques de base et les fonctionnalités offertes par les bibliothèques pour pouvoir les combiner de manière élégante et efficace.

Ce chapitre a donc pour but d'aller au delà des considérations syntaxiques vues dans les chapitres précédents, et de présenter quelques conseils et astuces permettant de réaliser des programmes plus fiables et plus sûrs. Il ne présentera bien entendu pas toutes les techniques qui peuvent être mises en place, tant ces techniques peuvent être diverses et variées, mais il essaiera de débroussailler un peu le terrain.

Les conseils donnés ici n'ont évidemment pas force de loi. Cependant, ils peuvent vous sensibiliser sur certains points dont peu de programmeurs débutants ont connaissance. Ils vous donneront peut-être aussi des idées et vous aideront sans doute à définir vos propres règles de codage. Vous êtes donc libres de vous en inspirer si vous ne voulez pas les appliquer telles quelles.

Nous verrons d'abord l'importance des conventions de codage. Quelques méthodes classiques permettant de réaliser des programmes fiables et évolutifs seront ensuite abordées. Enfin, nous présenterons les principales considérations système qu'il est préférable d'avoir à l'esprit lorsque l'on réalise un programme.

12.1. Conventions de codage

Une « convention de codage » est un ensemble de règles définissant le style et la manière de programmer. Le but des conventions de codage est souvent de garantir une certaine uniformité dans les codes sources produits par plusieurs personnes d'un même groupe, mais elles peuvent également permettre de réduire les risques de bogues de bas niveau.

12.1.1. Généralités

De nombreuses conventions de codage ont été définies. Quasiment chaque entreprise, voire même chaque groupe de développeurs, utilise une convention de codage qui lui est propre.

Les conventions de codage sont donc nombreuses, bien entendu non standardisées, et très souvent incompatibles. De plus, elles sont souvent très restrictives et imposent des règles qui n'ont pas toujours une autre justification que des considérations de style. Les règles de ces conventions relèvent donc de l'esthétique, et comme les goûts et les couleurs sont quelque chose de personnel, peu de programmeurs les considèrent comme une bonne chose.

Alors, ces conventions doivent-elles être ignorées ? Sans doute pas. Une mauvaise convention est toujours préférable à une anarchie du code, car elle permet au moins de fixer un style de programmation. Si l'on ne devait ne donner qu'une seule règle, ce serait sans doute de s'adapter à l'environnement dans lequel on se trouve, et de faire du code « mimétique ». En effet, l'anarchie ne va pas très bien avec la rigueur qu'un code doit avoir, et l'hétérogénéité des solutions choisies multiplie les risques de bogues et les dépendances des programmes.

> **Note :** Certains programmeurs ont une vision artistique de la programmation. Cela n'est pas contradictoire avec le respect de règles. Après tout, une démonstration mathématique bien articulée a également une certaine beauté. Mais l'essentiel est qu'à partir d'un certain niveau, seules les procédures et les règles de qualité peuvent garantir la fiabilité d'un programme.

En fait, outre la cohérence, les conventions de codage permettent de prévenir les bogues, en imposant une manière de travailler qui suit les règles de l'art.

En effet, contrairement à d'autres langages plus stricts et qui interdisent certaines fonctionnalités ou certaines constructions syntaxiques risquées, le C/C++ permet de réaliser virtuellement n'importe quoi. Il est donc très facile de faire des erreurs grossières en C/C++. Ces erreurs, bien que conduisant généralement le programme à une issue fatale, ne sont généralement pas des erreurs complexes. Cependant, elles peuvent être difficiles à localiser et à diagnostiquer a posteriori, même si, une fois diagnostiquées, elles peuvent être corrigées de manière extrêmement facile.

Or, la plupart de ces erreurs peuvent être évitées a priori en suivant des règles élémentaires d'hygiène de codage, pourvu que le programmeur s'y tienne. C'est pour cela que, comme le dit l'adage, mieux vaut prévenir que guérir, et respecter des règles de base justifiées par une argumentation pratique ne peut pas faire de mal.

En résumé, les bonnes conventions de codage s'intéressent plus au fond qu'à la forme. Elles ne définissent pas de règles contraignantes et inutiles, mais au contraire facilitent le développement et réduisent les coûts de mise au point, en éliminant les facteurs de bogues à la source et en augmentant la qualité du logiciel par une cohérence accrue.

Nous présenterons donc quelques-unes de ces règles et leur justification pratique dans les sections suivantes, en les classant suivant les principaux objectifs des conventions de codage.

12.1.2. Lisibilité et cohérence du code

La lisibilité et la cohérence du code source est une aide à sa compréhension, et facilite sa mise au point, sa maintenance et sa réutilisation. Elle peut également être une aide pour le programmeur lui-même, pour se retrouver dans son propre code source. Tout ce qui peut améliorer la lisibilité et la cohérence du code est donc une bonne chose, et les règles suivantes y contribuent directement.

12.1.2.1. Règles de nommage des identificateurs

La règle fondamentale pour faciliter la lecture du code est de s'assurer qu'il est auto-descriptif autant que faire se peut. Pour cela, il faut s'assurer que les noms des identificateurs soient le plus proche de leur sémantique possible, et éviter au maximum les conventions implicites.

N1. Le nom d'une fonction doit dire ce qu'elle renvoie.
N2. Le nom d'une procédure doit dire ce qu'elle fait.
N3. Le nom d'une variable doit dire ce qu'elle est ou l'état qu'elle représente.

Ces règles sont donc très importantes, car elles permettent de garantir que l'on sait exactement ce à quoi sert une entité à la simple lecture de son nom. Ainsi, le risque de mal utiliser cette entité est réduit.

Bien entendu, ces règles deviennent vitales dans le cas de code partagé entre plusieurs projets ou plusieurs programmeurs, ou dans le cas de code de bibliothèque susceptible d'être réutilisé dans un contexte a priori inconnu.

Exemple 12-1. Noms autodescriptifs

```
unsigned get_best_score();
int connect_to_http_server(const char *);
```

N4. Nommer les entités suivant la sémantique de leur implémentation et non suivant la fonctionnalité fournie.

Cette règle est un corollaire des règles précédentes. Elle permet de garantir une bonne utilisation de l'entité en question. Elle implique, en cas de changement d'implémentation susceptible de modifier le comportement, un renommage. Cela est le meilleur moyen pour détecter facilement les utilisations qui deviennent incompatibles suite à ce changement.

Inversement, un nommage basé uniquement sur la fonctionnalité fournie est susceptible de provoquer une dégradation des performances, voire des effets de bords si plusieurs fonctionnalités sont combinées de manière incompatibles du point de vue implémentation.

Par exemple, si une classe doit être définie spécifiquement pour gérer une file d'objet, il est préférable de lui donner un nom explicite quant à ses capacités réelles plutôt qu'un nom abstrait représentant la fonctionnalité obtenue :

```
class CQueue            // Imprécis, peut être utilisé à mauvais escient.
{
    // Implémentation quelconque.
    ...
public:
    void add_tail(object_t *);
    object_t *get_head();
    object_t *search(const char *name);
};

class CList             // Précis, on ne l'utilisera pas si on a besoin
{                       // de rechercher un élément.
    // Implémentation basée sur une liste.
    ...
public:
    void add_tail(object_t *);
    object_t *get_head();
    // Recherche possible, mais fatalement lente car basée
    // sur un parcours des éléments de la liste :
    object_t *search(const char *name);
};
```

> **Note :** Cette règle est facultative si l'encapsulation des fonctionnalités est parfaite et l'abstraction totale. Cependant, réaliser une abstraction totale des structures de données est une chose rarement réalisable ou même désirable, car cela conduit généralement à une implémentation structurellement mauvaise ou inadaptée au problème à résoudre. Voir les règles d'optimisation pour plus de détails à ce sujet.
>
> Cette règle est donc surtout importante pour les bibliothèques utilitaires génériques, qui fournissent des fonctionnalités dont on ne peut prévoir a priori le cadre d'utilisation. Il est dans ce cas préférable de nommer les entités relativement à leur implémentation et de laisser au programmeur le choix des fonctionnalités et des classes utilisées en fonction de l'utilisation qu'il veut en faire.

N5. En cas de surcharge, conserver la sémantique initiale de la méthode surchargée.

Cette règle est impérative pour préserver les règles précédentes. Elle est essentielle pour éviter les effets de bords.

Par exemple, si l'on redéfinit un opérateur d'addition pour une classe implémentant un type de données, il est essentiel que l'opération fournie soit en rapport avec la notion d'addition ou de regroupement. Cela vaut bien entendu aussi pour toutes les méthodes surchargeables des classes de base : il faut en conserver la sémantique à tout prix.

N6. Utiliser la notation hongroise simplifiée pour les variables et des données membres.

La notation hongroise simplifiée propose de qualifier les variables et les données membres d'un préfixe permettant d'en décrire la nature et le type. Cela permet de déterminer le type d'une variable sans avoir à rechercher sa déclaration, et de détecter directement à la lecture du code des erreurs de base. Il s'agit donc d'une technique complémentaire aux règles de nommage descriptives précédentes, qui permet de décrire la nature des entités manipulées et non plus seulement ce qu'elles sont.

De nombreuses variantes de cette notation sont utilisées de part le monde, parce que les préfixes ne sont souvent pas uniques et laissent une part d'ambiguïté, qui est en général levée par le contexte du code. Cette technique n'est donc pas parfaite, mais elle est préférable à aucune caractérisation de la nature des variables.

Le tableau suivant vous donnera un jeu de préfixes possibles, directement dérivés des noms anglais des types de données et choisis pour minimiser les conflits entre les préfixes.

Tableau 12-1. Préfixes en notation hongroise simplifiée

Type	Préfixe
Booléens	b
Caractères simples	c
Caractères larges	wc
Entiers courts	h
Entiers natifs	i
Entiers longs	l
Taille (type size_t)	s
Type intégral non signé	u
Type intégral signé	s
Énumérations	e
Flottants en simple précision	f
Flottants en double précision	d
Chaîne C simple	sz
Chaîne C en caractères larges	wsz
Tableaux	a
Pointeurs	p

Les préfixes doivent être combinés pour caractériser les types complexes. L'ordre généralement adopté pour les préfixes est le suivant : préfixe des tableaux ou des pointeurs, préfixe du signe, préfixe de taille et préfixe du type de données.

Exemple 12-2. Notation hongroise simplifiée

```
unsigned int uiItemCount = 0;
long int lOffset = 0;
const char *szHelloMessage = "Hello World!";
bool bSuccess = false;
void *pBuffer = NULL;
double adCoeffs[256];
```

N7. Préfixer les données membres par « `m_` », les variables globales par « `g_` », et les constantes par « `k_` ».

Cette règle permet de garantir qu'il n'y a pas de conflit entre des variables homonymes de portées différentes. Elle contribue donc à limiter les effets de bords implicites et permet au programmeur de savoir en permanence ce qu'il manipule.

```
const unsigned k_uiMaxAge = 140;    // Constante.

class CMan
{
private:
    unsigned m_uiAge;               // Donnée membre
    ...
};

CPeople g_World;                    // Variable globale.
```

12.1.2.2. Règles de cohérence et de portabilité

Les règles de cohérence permettent d'obtenir un code source plus facile à lire, à maintenir et à faire évoluer. En effet, ces règles permettent de savoir exactement comment sont nommées les entités du programme, et donc évitent d'avoir à rechercher leur définition en supprimant tout doute sur son nom. Il est à noter que la plupart des programmeurs finissent par acquérir de manière inconsciente un jeu de règles de cohérence, et qu'ils considèrent leurs anciens programmes comme non structurés après cela.

En général, ce n'est pas la formalisation des règles de cohérence qui prime, mais le fait même qu'elles existent. De ce fait, les règles données ci-dessous n'imposent aucun choix, mais servent seulement à donner une idée de ce qui peut être pris en compte pour garantir la cohérence du code source.

Il faudra toutefois veiller à ne pas définir de règles trop nombreuses ou trop spécifiques, afin de ne pas encombrer l'esprit du programmeur et risquer ainsi de réduire sa productivité ou, pire, de lui faire rejeter l'ensemble des règles en raison de leurs trop fortes contraintes. Ici donc, tout est affaire de mesure, d'autant plus que la finalité de ces règles n'est pas directe.

C1. Être cohérent et consistant dans la forme des noms des identificateurs.

Les noms d'identificateurs doivent avoir toujours la même forme. Généralement, ils sont constitués de verbes et de noms. On peut imposer l'emploi d'un verbe et la manière de le conjuguer pour les procédures par exemple. De même, les mots utilisés dans les identificateurs peuvent être séparés par des soulignements bas (caractère '_'), ou tout simplement accolés mais identifiés par une majuscule en tête de chaque mot.

Généralement, définir comment accoler ces noms suffit. Les deux conventions les plus utilisées sont :

- soit de coller les mots et de les mettre en majuscules (par exemple `CreateFileMapping`), sauf éventuellement la première lettre ;
- soit de les mettre en minuscules et de les séparer par des soulignements (par exemple `sched_get_priority_max`).

C2. Éviter les redéfinitions de types fondamentaux.

Nombre de programmeurs ont ressenti le besoin de redéfinir les types fondamentaux, en raison de leur manque de portabilité. Parfois, la raison est la recherche d'une abstraction vis à vis du type de données choisie. Cette technique est à éviter à tout prix, pour trois raisons :

- premièrement, des types portables sont à présent disponibles dans l'en-tête `stdint.h` ;
- deuxièmement, le choix d'un type est un choix de conception sur lequel on ne doit pas avoir à revenir, et abstraire le type des instances est généralement inutile ;
- enfin, les programmes finissent par manipuler une foison de types a priori identiques, qui ne servent à rien, et que l'on doit convertir de l'un à l'autre en permanence.

Exemple 12-3. Exemple de types redondants

```
guint32   // Inutile, utiliser uint32_t.
UINT      // Inutile, utiliser unsigned.
uint      // Inutile, utiliser unsigned.
DWORD     // Inutile et imprécis, utiliser uint32_t.
```

La situation peut devenir encore plus grave avec les types plus complexes comme les types de chaînes de caractères (LPWSTR, BSTR, LPOLESTR, CString, CComBSTR, bstr_t, wstring, etc. sous Windows par exemple), car la conversion n'est dans ce cas pas immédiate. Beaucoup de programmes finissent ainsi par faire des conversions en série, ce qui grève fortement les performances.

C3. Ne pas mélanger les types de gestion d'erreur.

Entre les exceptions, les codes de retour et les `goto`, il faut faire son choix. Mélanger les techniques de gestion d'erreur conduit directement à du code complexe, illisible et tout simplement non fiable (voir la Chapitre 8 pour plus de détails à ce sujet).

Respecter cette règle n'est pas toujours facile, surtout si l'on utilise des bibliothèques de provenances différentes. Dans ce cas, le plus simple est d'appliquer la règle en vigueur dans le programme principal, dont le code source est généralement majoritaire.

C4. Fixer les codes d'erreurs et les classes d'exception globalement.

Une fois la technique de gestion des erreurs définie, il est essentiel de définir une politique globale de définition des erreurs. Cette politique est particulièrement importante si la gestion des erreurs se fait via des codes de retour, car dans ce cas il peut y avoir conflit entre plusieurs programmeurs. Dans le cas des exceptions, les conflits sont évitables, mais le traitement des erreurs génériques via des exceptions polymorphiques peut induire des erreurs inattendues si une classe d'exception n'est pas bien située dans la hiérarchie des exceptions.

C5. Fixer la langue de codage.

Les noms d'identificateurs doivent être soit en anglais (recommandé pour les programmes à vocation internationale ou Open Source), soit dans la langue maternelle du développeur, mais jamais dans un joyeux mélange des deux. Dans le cas contraire, la situation peut devenir catastrophique, surtout pour

pour les codes d'erreurs utilisés couramment ou les fonctions de bibliothèque.

L'exemple suivant présente quelques cas de franglais relativement classiques :

```
create_fichier          // Franglais.
ERROR_SATURE            // Franglais.
IsValideFilePath        // Faute d'orthographe anglaise.
```

Souvent, les développeurs non anglophones ne parviendront pas à respecter cette règle si l'anglais est choisi, ou feront des fautes d'anglais énormes qui ne seront pas du meilleur effet. Si le programme est à vocation locale, autant dans ce cas ne pas les ennuyer et choisir la langue maternelle. La lecture du programme peut toutefois en pâtir, en raison du fait que les mots clés resteront toujours en anglais (bien entendu, il est absolument interdit de définir des macros localisées pour chaque mot clé du langage !).

C6. Utiliser des noms de fichiers en minuscules et sans espaces.

Certains systèmes ne gèrent pas la casse des noms de fichiers. Il est donc impératif de fixer une règle, car les programmes qui n'en fixent pas ne seront dès lors pas portables. En effet, le transfert d'un code source fait sous Unix vers Windows peut ne pas être réalisable si les fichiers sont homonymes sur le système cible. Inversement, des fichiers sources provenant de Windows ne seront a priori pas trouvés par le préprocesseur dans les directives #include si la casse n'est pas la même dans la directive et dans le nom de fichier (chose que les préprocesseurs pour Windows ne vérifient bien sûr pas).

```
#include "Mauvais Nom.h"    // Non portable (un espace et casse variable).
```

C7. Utiliser le séparateur '/' dans les directives `#include`.

Certains systèmes d'exploitation utilisent le séparateur '`\`' pour les noms de fichiers. Utiliser ce caractère est non portable. Le caractère '`/`' étant compris par tout préprocesseur standard, et ce quel que soit le système utilisé, autant n'utiliser que lui. De plus, en C, le caractère '`\`' doit être doublé dans les chaînes de caractères, y compris dans les directives `#include`. Cela peut généralement être évité sur les systèmes qui ont bidouillé leur préprocesseur, mais le faire provoquera immanquablement une erreur de compilation sur les systèmes conformes.

```
#include "utilities\containers.h"       // Incorrect
#include "utilities\\containers.h"      // Toujours incorrect
#include "utilities/containers.h"       // Correct
```

C8. Faire des en-têtes protégés.

Un fichier d'en-tête doit pouvoir être inclus plusieurs fois. Il faut donc utiliser les directives de compilation conditionnelle pour éviter des erreurs dues à des déclarations multiples (surtout en C++, la structure des classes étant définie dans les fichiers d'en-tête).

On n'utilisera pas les mécanismes propriétaires non portables, du type « `#pragma once` » (ils ne font gagner que deux lignes par fichier au prix d'un non-respect des standards). On remplacera systématiquement ceux ajoutés automatiquement par les générateurs de code conçus pour rendre les programmes dépendants d'une plateforme spécifique.

```
#ifndef __MONPROJET_FICHIER_H__
#define __MONPROJET_FICHIER_H__

// Contenu du fichier d'en-tête
```

```
#endif // __MONPROJET_FICHIER_H__
```

12.1.2.3. Règles de formatage et d'indentation

Les règles d'indentation et de forme relèvent du style. Elles sont soumises aux mêmes contraintes que les règles de cohérence, à laquelle elles contribuent. Autrement dit, elles ne devront pas être contraignantes, et les choix qu'elles imposent sont moins importants que le fait qu'elles existent.

F1. Indenter le code.

L'indentation est une aide à la lecture et au codage. Elle permet également de voir plus rapidement la structure du programme. Cependant, les styles de codage, ainsi que les caractères utilisés pour indenter, varient grandement. La règle est donc ici de se plier aux conventions utilisées pour le projet sur lequel on travaille.

En particulier, les caractères à utiliser pour l'indentation peuvent être des espaces ou des tabulations. Certaines personnes prétendent que seuls les espaces doivent être utilisés pour indenter, car la taille des tabulations n'est pas fixe et varie selon les éditeurs de texte. De plus, les tabulations font souvent huit caractères sur les imprimantes texte, ce qui empêche d'imprimer le code source correctement.

Cependant, le caractère de tabulation a clairement pour but de réaliser un décalage d'une colonne, et il n'y a par conséquent aucune raison de ne pas l'utiliser pour cela. Prétendre que cela est gênant pour des raisons d'outils n'est pas honnête de nos jours, car tous les outils permettent depuis longtemps de paramétrer la taille des tabulations (une taille de quatre caractères est généralement pratiquée) et d'imprimer correctement un listing.

Quoi qu'il en soit, quel que soit le choix effectué, il faut s'y tenir sur l'ensemble des fichiers sources : dès que l'on utilise des espaces, il ne faut utiliser que cela pour les indentations (et inversement). Dans le cas contraire, l'indentation ne sera pas la même pour les lignes qui utilisent des tabulations et celles qui utilisent des espaces, et si plusieurs programmeurs travaillent sur un même projet avec des tailles de tabulation différentes, le code ne sera pas lisible sans reconfigurer les éditeurs pour chaque fichier source.

F2. Commenter le code.

Sans commentaire. Toutefois, on prendra conscience du fait que les commentaires utiles sont préférables aux commentaires tautologiques, et qu'il faut donc s'efforcer de faire des commentaires clairs et concis. Généralement, on ne commente pas un code déjà clair et simple, même si cela n'est pas interdit.

En revanche, il est important de commenter les structures de données, les fonctions et méthodes. Cela doit se faire de préférence dans les fichiers d'en-tête, car c'est ce fichier qui constitue la déclaration de l'interface de programmation, et souvent le seul fichier source distribué dans le cas des bibliothèques.

Le formalisme des commentaires pour ces fonctions et méthodes doit permettre de les utiliser sans risque. Les informations importantes sont tout particulièrement les suivantes :

- les paramètres en entrée ;
- les paramètres en sortie ;
- ce que renvoie la fonction ou ce que fait la méthode ;
- les codes de retours, exceptions et autre cas d'erreurs ;
- les préconditions (dans quelles circonstances ou quel ordre les fonctions doivent être appelées) ;
- les effets de bord et les remarques complémentaires.

Exemple 12-4. Commentaire descriptif d'une fonction

```
/*
        Normalise un vecteur.
    Entrée :
        vector :   Pointeur sur le vecteur devant être normalisé.
    Sortie :
        pResult :  Vecteur normalisé.
        pNorm :    Norme du vecteur avant normalisation.
    Retour :
        0 si succès,
        -1 si l'un des pointeurs de sortie est nul,
        -2 si le vecteur à normaliser est nul.
    Précondition :
        Aucune
    Effets de bords :
        Aucun
    Description :
        Le vecteur normalisé est le vecteur colinéaire au vecteur d'entrée
    et dont la norme est 1. Il est calculé en divisant chaque composante
    du vecteur à normaliser par sa norme. Ce calcul est impossible
    si le vecteur en entrée est le vecteur nul.
*/
extern int get_normalize(const vector_t &vector,
    vector_t *pResult, double *pNorm);
```

On notera que des outils performants permettent d'extraire la documentation à partir du code. Les documents générés par ces outils ne peuvent toutefois pas se substituer à une documentation complète. En particulier la documentation présentant les concepts généraux d'un programme ou d'une bibliothèque, ainsi que les documents de conception, qui doivent être réalisées avant le code de toutes manières, restent nécessaires.

F3. Commenter les `#endif`.

Les `#if` et `#endif` ne se laissent pas indenter facilement. De ce fait, ils apparaissent à plat dans les codes sources. Il est donc utile de commenter chaque `#endif` en rappelant la condition du `#if` correspondant.

F4. Utiliser les espaces dans les ASCII art.

Il est possible de faire un schéma en « ASCII art » dans un code source. Dans ce cas, il ne faut surtout pas utiliser le caractère de tabulation pour aligner les différentes parties du schéma. En effet, la largeur de ce caractère dépend de l'outil utilisé pour visualiser le code source. De ce fait, il ne faut utiliser que des espaces pour les blancs dans les schémas de ce type.

12.1.3. Réduction des risques

De nombreuses erreurs de bas niveau peuvent être évitées dès la phase de codage simplement en respectant quelques règles. Les règles suivantes permettent donc d'augmenter la fiabilité du code et de faire gagner du temps pendant la phase de mise au point.

12.1.3.1. Règles de simplicité

Rechercher la simplicité au sens général est la règle d'or. Les anglophones disent « KISS » (abréviation de « Keep It Simple Stupid »), pour bien traduire le fait que l'être humain est limité et n'est pas capable de comprendre les choses complexes. De plus, les choses ont suffisamment tendance à se compliquer toutes seules pour que le programmeur n'en rajoute pas.

S1. Utiliser des algorithmes simples.

La complexité d'un algorithme est un passeport direct pour les erreurs, les effets de bords et les comportements non prévus. En effet, le comportement d'un programme peut très vite devenir combinatoire, et il est facile d'obtenir des algorithmes moyennement complexes ne pouvant déjà plus être testés.

S2. Éviter les astuces.

Les astuces de programmation ne plaisent qu'à celui qui les fait. Les autres perdent leur temps à analyser le code pour savoir ce qu'il fait. Elles sont donc à proscrire, d'autant plus qu'elles constituent un facteur de risque inutile.

S3. Ne pas faire d'optimisations locales.

Les compilateurs actuels sont parfaitement capables de faire les optimisations locales. Le programmeur qui fait de telles optimisations perd donc son temps et accroît les risques de bogues. Au pire, il peut même dégrader les performances, en trompant le compilateur sur ses véritables intentions et en le mettant dans une situation où il ne peut plus appliquer d'autres optimisations plus efficaces.

Enfin, les optimisations les plus efficaces sont généralement structurelles. Voir la section Optimisations à ce sujet.

12.1.3.2. Règles de réduction des effets de bords

Les effets de bords sont inhérents et même nécessaires aux langages impératifs, puisque ceux-ci se basent sur la modification de l'état du programme pour l'exécuter. Cependant, il faut distinguer les effets de bords désirés (le traitement que le programme doit faire) des effets de bords indésirés (les bogues). Les règles suivantes ont donc pour but d'éviter les constructions à risques et de cloisonner les données accessibles aux traitements qui en ont besoin.

B1. Éliminer les variables globales.

Les variables globales sont accessibles de l'ensemble du programme et le risque qu'elles soient modifiées par inadvertance ou dans le cadre d'un effet de bord non documenté d'une fonction ou d'une procédure est maximum. Or, dans la majorité des cas, une variable n'a pas besoin d'être globale et son accès peut être réduit à une portion de code réduite.

Il est donc impératif de réduire au maximum la portée des variables et de les déclarer au plus proche de leur utilisation. Cela implique de déclarer les variables le plus proche possible du bloc d'instructions qui en a besoin, voire, en C++, à l'endroit même où elle est utilisée (et non au début du bloc d'instructions).

De même, les données manipulées par les fonctions et les méthodes devront, autant que faire se peut, être fournies en paramètre et non accédées directement. Cela permet également de rendre ces méthodes et ces fonctions autonomes et utilisables de manière indépendante de tout contexte, facilitant ainsi leur réutilisation et leurs tests unitaires.

B2. Utiliser les objets.

La notion d'objet permet de réduire la portée des variables en les encapsulant et en les regroupant

avec le code qui les utilise, et de fournir l'accès à ces variables implicitement dans les méthodes de la classe via le pointeur sur l'objet. Le principal avantage de la programmation objet est donc sans doute, avec la structuration du code, de permettre la réduction de l'utilisation des variables globales. Il ne faut donc pas s'en priver.

On gardera toutefois à l'esprit qu'une donnée membre d'une classe n'est rien d'autre qu'une variable globale pour toutes les méthodes de cette classe. La règle de localité s'applique donc également aux données membres, et si une méthode est la seule à utiliser une donnée ou si cette donnée peut être fournie en paramètre sans préjudice de la lisibilité du code, alors cette donnée doit être déclarée localement et non en tant que données membre.

B3. Ne pas exposer la structure interne.

S'il est important de conserver à l'esprit la nature de l'implémentation des classes et des fonctions que l'on utilise, leur implémentation elle-même doit être inaccessible autant que faire se peut. En effet, laisser un accès incontrôlé à l'implémentation d'une fonctionnalité est une manière de rendre globale sa structure ou son mécanisme. De plus, c'est un frein aux évolutions ultérieures du programme, car la modification de l'implémentation ne peut plus se faire de manière aussi facile si l'ensemble du code s'appuie sur des détails de cette implémentation.

Les techniques d'encapsulation sont nombreuses, et la programmation objet permet de les mettre en œuvre facilement. En particulier, on s'assurera que les droits d'accès aux données membres sont minimaux et que, pour les classes les plus importantes, les fonctionnalités offertes sont structurées et exposées via la notion d'interface.

B4. Éviter les encapsulations à effets de bord.

Les interfaces et les accesseurs fournis doivent impérativement éviter de réaliser des opérations susceptibles d'avoir d'autres effets que les effets documentés ou déductibles des noms de leurs identificateurs. Autrement dit, ils ne doivent en aucun cas avoir d'effets de bords. Cette règle est un corollaire des règles de nommage et de réduction des risques déjà vues, appliquées aux techniques objet.

Par exemple, les accesseurs en lecture ne doivent pas modifier l'état de l'objet auxquels ils s'appliquent. Les accesseurs en écriture sont bien entendu obligés d'avoir des effets de bords, mais ceux-ci sont documentés et connus de celui qui les utilise.

Exemple 12-5. Accesseur à effet de bord

```
int CTimer::GetCurrentValue()
{
    NotifyClients();    // Effets de bords incontrôlables !
    return m_iValue;
}
```

B5. Utiliser const.

Une manière simple de réduire les effets de bords est d'utiliser le mot clé const. Cela permet de garantir, via les mécanismes de typage du langage, que les données qui ne doivent pas être modifiées ne le seront pas par inadvertance.

De plus, les compilateurs sont souvent capables d'optimiser la manipulation des données dont ils savent qu'elles ne peuvent être modifiées par le code qui les utilise. Ils n'ont pas à maintenir un certain nombre de règles de cohérence et peuvent donc appliquer des optimisations plus poussées. Ce peut être le cas par exemple pour le passage par valeur de types complexes : un passage par valeur constante peut souvent être transformé en un passage par référence constante et éviter ainsi des copies coûteuses d'objets.

B6. Passer les paramètres de retour par pointeur.
B7. Passer les paramètres d'entrée par référence constante ou par valeur.

Les paramètres de retour des méthodes et des fonctions doivent être clairement identifiés comme tels. Pour cela, il est conseillé d'utiliser des pointeurs, forçant ainsi le programmeur à prendre conscience du fait que les variables dont il fournit l'adresse seront modifiées.

L'utilisation des références pour les paramètres de retour est déconseillée, car elle ne permet pas de distinguer, lors de l'écriture de l'appel de la méthode, les variables qui sont susceptibles d'être modifiées de celles qui ne le seront pas. Il est donc facile, avec les passages de paramètres par référence, d'obtenir des effets de bords indésirés.

De la même manière, les paramètres d'entrée devront être passés soit par valeur, soit par référence constante s'ils sont de grande taille. En aucun cas ils ne devront être passés par référence non constante, car cela permettrait de modifier les données fournies par l'appelant.

Enfin, les paramètres d'entrée / sortie peuvent, exceptionnellement, être passés par référence, si la sémantique de la fonction ou de la procédure est suffisamment claire pour qu'il n'y ait pas d'ambiguïté sur le fait que ces paramètres peuvent être modifiés lors de l'appel.

Exemple 12-6. Passage de paramètres en entrée/sortie

```
void get_clipping_rectangle(const window_t &window, rect_t *pRect);
int set_clipping_rectancle(window_t &window, const rect_t &rect);
```

B8. Éviter les macros.

Les macros sont, par définition, un moyen de réaliser plusieurs opérations avec une écriture simplifiée. De ce fait, elles sont susceptibles de réaliser des effets de bords de manière très simple. De plus, les macros sont difficilement débogables, peuvent évaluer plusieurs fois leurs paramètres d'entrée, et ne sont pas soumises aux contrôles de vérification des types du langage. Il est donc important d'éviter au maximum les macros, et de les réserver uniquement à la définition de constantes utilisées dans les directives de compilation conditionnelle. N'oubliez pas que les macros peuvent souvent être avantageusement remplacées par des fonctions inline.

12.1.3.3. Règles de prévention

Un code peut être utilisé dans des conditions qui n'étaient pas connues lors de sa conception, même en utilisation nominale, car les programmes sont des entités complexes dans lesquels l'ensemble des choix de l'utilisateur peut rarement être prévu. De plus, un programme peut être appelé à évoluer et à subir des modifications du contexte d'utilisation de son code source. Le code source peut également être récupéré dans un autre programme, le mettant dans une situation a priori non prévue par son développeur.

Ce genre de situation peut amener à des erreurs dues à des hypothèses implicites ou à des impasses qui ont été faites lors du codage initial. Les règles suivantes permettent de garantir qu'un code restera fiable en toute circonstance, principalement en fixant le contexte et en forçant le programmeur à prévoir les chemins détournés.

P1. Fixer et documenter les conventions d'appel.

Les conventions d'appel sont les conventions qui décrivent la manière dont les méthodes et les fonctions doivent être appelées. Il existe par exemple des conventions d'appel pour chaque langage ou chaque système, afin de décrire les mécanismes utilisés pour effectuer les passage des paramètres (ordre de passage des paramètres, types de données natifs utilisés, qui de l'appelé ou l'appelant doit se charger de la destruction des paramètres, etc.). Mais il est également possible de définir des conven-

tions d'appel de plus haut niveau, qui seront utilisées pour tout un programme.

Ces conventions doivent être définies et appliquées globalement, tout comme les conventions de codage, pour être efficaces. Elles doivent au minimum décrire les mécanismes d'allocation mémoire utilisés lorsque des blocs mémoires doivent être transférés d'une fonction à une autre, et la manière de remonter les erreurs. Les valeurs de codes de retour ne feront pas exception à la règle (afin d'éviter, par exemple, que la valeur 0 signale tantôt une erreur, tantôt un succès en retour de fonction).

Le problème de l'allocation mémoire est complexe. Supposons que l'on désire appeler une fonction qui retourne une chaîne de caractères. Cette chaîne doit être stockée dans une zone mémoire, dont la taille est a priori dépendante du résultat de la fonction. Il est donc courant d'utiliser une allocation dynamique de mémoire pour retourner le résultat. Mais si le code qui utilise cette fonction et la fonction elle-même utilisent des conventions d'appel différentes, le programme risque de faire des erreurs mémoire très grave et planter soit immédiatement, soit, pire encore, bien après l'utilisation de la fonction.

Par exemple, la fonction `strdup` de la bibliothèque C des systèmes compatibles Unix 98 permet de dupliquer une chaîne de caractères et d'en retourner la copie dans un tampon alloué par la fonction `malloc` :

```
// Duplication d'une chaîne de caractères :
char *szCopy = strdup("Hello World!");
```

La libération de la mémoire doit être réalisée avec la fonction `free`. Or, un programme C++ qui appellerait `delete[]` sur la chaîne copiée ne serait pas correct (bien que dans la plupart des cas, les opérateurs `new` et `delete` du C++ utilisent les fonctions de gestion de la mémoire de la bibliothèque C sous-jacente).

Les conventions d'appel doivent donc être définies de manière stricte, et si possible être uniformes dans tout le programme.

P2. Initialiser les variables.

Les variables non initialisées sont extrêmement dangereuses, car elles peuvent contenir n'importe quelle valeur. Lorsqu'elles sont utilisées dans un algorithme qui suppose que leur valeur est correcte, celui-ci adopte un comportement aléatoire, souvent non reproductible, et susceptible d'effectuer des traitements qui ne sont logiquement pas prévus par le programme. Cela conduit donc à des bogues difficiles à reproduire et à diagnostiquer, et capables de générer des effets de bords complexes.

Dans le cas des pointeurs, cette règle est absolument vitale, car l'utilisation d'un pointeur non initialisé peut conduire au mieux à un plantage immédiat, au pire à une écriture arbitraire dans la mémoire du programme !

L'initialisation d'une variable à une valeur par défaut, même invalide pour la sémantique du programme, est donc impérative. C'est une opération que l'on doit réaliser dès la création de la variable, et permet de garantir la reproductibilité du comportement du programme. Cette initialisation doit être réalisée même lorsque l'algorithme qui l'utilise permettrait de s'en passer, car cela relève de l'optimisation inutile et accroît les risques considérablement.

Toute définition de variable doit donc être immédiatement suivie de son initialisation, et tout ajout d'une donnée membre à une classe doit être immédiatement suivi de l'écriture de son code d'initialisation dans le constructeur ou dans le fichier d'implémentation pour les variables statiques. Ce comportement doit être acquis comme un simple réflexe pour tout programmeur qui se respecte.

```
int i=0;
void *pBuffer = NULL;    // VITAL !
```

P3. Réinitialiser les variables détruites.

En complément de la règle précédente, une variable qui a été utilisée et dont on ne se servira plus, mais qui reste accessible dans le reste du programme, doit toujours être réinitialisée à une valeur invalide pour la sémantique du programme dès que l'on n'en a plus besoin. En particulier, tout pointeur dont la mémoire a été libérée doit être immédiatement réinitialisé à sa valeur nulle. Dans le cas contraire, les cas d'erreurs présentés dans la règle précédente redeviennent possible après destruction de la variable.

```
delete[] pBuffer;
pBuffer = NULL;    // Réinitialisation immédiate !
```

P4. Valider les entrées.

Toute donnée provenant de l'extérieur du programme doit être considérée comme non sûre. Elles peuvent être incorrectes, corrompues, ou tout simplement trafiquées dans le but d'obtenir du programme un comportement différent de celui pour lequel il est prévu. Il est donc essentiel de vérifier la validité de ces données (ce qui suppose, bien entendu, que les formats d'échange et de fichiers soient conçus pour permettre cette vérification facilement).

Les données dont la validation devra être effectuée comprennent notamment :

- les données lues à partir d'un fichier ou récupérées via une connexion réseau ;
- les données fournies par l'utilisateur ;
- les données fournies au travers des interfaces publiques.

En revanche, il est inutile de valider les données fournies en paramètre à une fonction privée ou interne à un programme, car on peut supposer dans ce cas que le contexte d'appel est maîtrisé. Si l'on désire malgré tout faire des vérifications dans ce cas, il est préférable d'utiliser la macro `assert` (voir plus bas).

P5. Toujours coder default et else.

Les branchements conditionnels correspondent à différents cas d'utilisation d'un logiciel. Chaque branchement implique un choix, pour lequel il doit y avoir une réponse appropriée. Le fait de ne pas donner de réponse à l'un de ces choix constitue un bogue qui peut se produire ou non, selon que la situation considérée peut effectivement se présenter ou non.

Cependant, la possibilité qu'une situation se présente ou non est un facteur extérieur au programme, et même dans le cas de choix logiquement exclusifs, faire une impasse revient à considérer que la logique du programme ne changera jamais. Par conséquent, si l'on veut s'assurer qu'un programme fonctionnera toujours ou, au moins, signalera les situations pour lesquelles il ne peut plus assurer un comportement déterminé, il faut prévoir l'ensemble des possibilités dès le codage.

Cela se traduit en pratique par l'écriture systématique d'un `else` pour chaque `if`, et d'un cas par défaut pour chaque `switch`. Cette manière de procéder force le programmeur à se poser la question de la possibilité que la condition inverse puisse se produire. Bien entendu, dans de nombreux cas, cette condition ne nécessitera aucun traitement particulier. L'utilisation de l'instruction vide peut alors simplement permettre de montrer que ce cas de configuration a bien été pris en compte par le programmeur.

```
if (i>10)
    i = 10;
else
    ;        // On accepte les nombres négatifs.
```

P6. Utiliser `assert`.

La macro `assert` (déclarée dans le fichier d'en-tête `assert.h`) permet de vérifier une condition dont la véracité doit toujours être assurée lors de son exécution. Elle prend en paramètre l'expression de la condition à vérifier, l'évalue et interrompt le programme si cette expression est fausse. Ce comportement est en effet la meilleure des choses à faire lorsque le programme se trouve assurément dans une situation non prévue.

La macro `assert` est donc un outil de débogage puissant, qui permet de garantir qu'un programme n'aura pas un comportement non prévu. Elle permet également de signaler la condition qui n'est plus vérifiée à l'arrêt du programme, permettant ainsi au programmeur de diagnostiquer l'erreur plus facilement. Son usage est donc particulièrement recommandé.

En pratique, `assert` sera utilisé dans toutes les situations où un test n'est pas fait car supposé comme toujours vrai, ou lorsqu'une branche du programme ne doit jamais être atteinte. Par exemple, les paramètres d'une fonction ou procédure interne peuvent être validés avec `assert` afin de détecter les erreurs des fonctions appelantes. De même, le cas par défaut d'un `switch` dont tous les cas ont été traités explicitement peut contenir un `assert` pour signaler qu'un cas a été oublié.

Exemple 12-7. Utilisation de assert

```
switch (eColor)
{
case color_red:
    do_red();
    break;
case color_green:
    do_green();
    break;
case color_blue:
    ;
    break;
default:
    assert(false);
    break;
}
```

P7. N'avoir qu'un seul return par fonction.

Toute fonction et toute procédure ne doivent avoir qu'un seul point de sortie. En effet, le fait d'utiliser return dans le corps d'une fonction ou d'une procédure a pour conséquence de masquer la sortie du flux d'exécution de manière prématurée. De ce fait, toute modification ultérieure de la fonction risque d'être réalisée en supposant que la fin de la fonction sera exécutée, alors que cela peut ne pas être le cas. Il peut s'ensuivre des erreurs graves dans la logique du programme, et généralement des consommations de ressources ou des interblocages dus au fait que le code de libération des ressources n'est pas exécuté dans des cas particuliers difficiles à reproduire.

```
int f(int i)
{
    lock();         // Prise de ressource.
    if (i < 2)
    {
        do_job();
        return 0;   // Dangereux, on oublie facilement le unlock() !
    }
    do_another_job();
```

```
    unlock();       // Libération de la ressource.
    return 1;
}
```

Note : En réalité, le seul cas d'utilisation valide de `return` dans le corps d'une fonction est tout au début de la fonction, pour sortir immédiatement en cas de détéction de paramètres incorrects dans le code de vérification des paramètres.

P8. Mettre les lvalues à droite dans les tests d'égalité.

Les « *lvalues* » (abréviation de « left values ») sont les expressions que l'on peut placer à gauche des opérations d'affectation. De ce fait, ce sont des expressions qui représentent une variable ou une référence de variable.

Du fait que l'opération d'affectation renvoie une valeur, les écritures telles que celles-ci sont tout à fait valides :

```
if (i = 2)
{
    // Toujours exécuté, car 2 est vrai.
    // De plus, i a perdu sa valeur d'avant le test.
}
```

Cela constitue généralement une erreur, car les tests vérifient généralement une condition et pas la non nullité d'une valeur affectée. De plus, ce type d'erreur se produit facilement, puisqu'il suffit d'une simple faute de frappe (un oubli de '=' en l'occurrence).

Par conséquent, il est recommandé de toujours placer les lvalues (`i` dans notre exemple) à droite dans les tests d'égalité :

```
if (2 == i)
{
    // Si i vaut deux, alors...
}
```

Dans le cas d'un oubli d'un '=', le compilateur signalera cette fois une erreur, car on ne peut affecter une valeur à une autre valeur.

P9. Faire des constructeurs et des destructeurs sûrs.

En raison de la grande difficulté de la gestion des erreurs dans les constructeurs et de son impossibilité dans les destructeurs, il est recommandé de ne faire que des opérations extrêmement simples dans ces méthodes. En particulier, le constructeur doit en général se limiter à l'initialisation des données membres non statiques de la classe.

Cette règle est particulièrement importante pour les classes qui peuvent être instanciées globalement. En effet, les objets globaux sont instanciés très tôt dans la durée de vie du programme, avant l'appel de la méthode `main`, et dans certains cas les ressources systèmes ne sont pas toutes accessibles. Un constructeur complexe peut échouer, et provoquer ainsi une erreur fatale difficile à diagnostiquer au lancement du processus.

En général, les opérations relatives à la gestion de la durée de vie des objets de classes complexes sont

affectées à des méthodes dédiées à ces tâches. Par exemple, une méthode `Init` peut être définie pour l'initialisation et une méthode `Reset` pour la destruction et la libération des ressources. De même, les opérations de copie d'une classe sont souvent identiques entre les constructeurs de copie et les opérateurs d'affectation, aussi l'implémentation d'une méthode `Clone` dédiée à cette tâche peut-elle être utile.

12.1.4. Optimisations

Sans précautions particulières, il est très facile de produire un code source qui n'est pas efficace ou qui est extrêmement lourd alors que cela n'est pas nécessaire. Pourtant, les règles suivantes permettent souvent de prévenir une bonne partie des problèmes de conception de bas niveau. Elles ont généralement pour principe de forcer le programmeur à se poser les bonnes questions lors de la conception ou lors d'un choix technique.

12.1.4.1. Règles d'optimisation générales

Généralement, les optimisations les plus efficaces sont les optimisations structurelles. De plus, les structures de données inadaptées à un problème provoquent souvent un code plus complexe, donc moins lisible et moins fiable.

Comme ce sont aussi les modifications structurelles qui sont les plus coûteuses dans un programme, il est important de bien réfléchir à la structure d'un programme dès le début du projet. Les règles suivantes ont donc principalement pour but de mettre en valeur l'importance de la structure et d'éviter les pièges de conception qui peuvent conduire à une structure de données inadaptée.

O1. Préférer les optimisations structurelles aux optimisations locales.

Comme il l'a déjà été indiqué, les optimisations les plus efficaces sont les optimisations structurelles. Inversement, les optimisations locales induisent souvent une complexité accrue et l'usage d'astuces qui nuisent à la lisibilité du programme et multiplient les risques de bogues.

Il est donc particulièrement important de réfléchir aux structures de données et à la conception globale d'un programme. Une bonne conception permet de garantir que les informations seront accessibles simplement et rapidement, rendant ainsi le programme efficace et le code source lisible et compréhensible.

O2. Analyser les compromis complexité / temps / taille et choisir les structures de données en fonction de l'usage.

Les choix de conception, notamment au niveau des structures de données et des algorithmes qui les manipulent, impliquent souvent des compromis. Généralement, il faut choisir entre simplicité des algorithmes, vitesse d'exécution et consommation mémoire.

Le choix doit se faire en fonction des objectifs recherchés du programme et de la nature des informations manipulées et de leur nombre. Ne pas prendre en compte ces informations peut conduire à un programme extrêmement consommateur de ressources et très lent. De plus, si les données utilisées ne sont pas accessibles facilement, le code source du programme sera plus complexe et les principes d'encapsulation seront plus facilement violés.

Il est donc particulièrement important de bien connaître les avantages et les inconvénients des différentes associations (associations, listes, tableaux, etc.) et de les utiliser à bon escient, en fonction du but à atteindre et des cas d'utilisation pratiques. Les choix structurels et les algorithmes qui en découlent seront bien entendu documentés afin de donner une vue d'ensemble du programme sans avoir à parcourir l'ensemble du code source pour en comprendre les mécanismes.

O3. Ne pas faire d'abstractions contraires à la réalité du problème.

Les abstractions et les généralisations permettent de réaliser du code générique et donc la factorisation du code et la réduction des risques de bogues. Toutefois, les abstractions ne doivent pas devenir un objectif en soi et ne doivent être réalisées que pour servir la cause du programme : résoudre un problème donné. Dans le cas contraire, la structure logique du programme sera peut-être élégante, mais elle ne conviendra pas pour la résolution du problème. De ce fait, les algorithmes utilisés, et donc le code au final, devront aller à l'encontre de cette structure, devenant ainsi complexes, difficilement maintenables, et risquant de violer en permanence les encapsulations en cherchant à obtenir des fonctionnalités non prévues pour les objets manipulés.

O4. Considérer la dérivation comme une agrégation.

Un cas particulier important de la règle précédente est de bien prendre conscience qu'un héritage, en C++, est une agrégation de la structure de données de la classe de base avec celle de la classe dérivée. Ce n'est en aucun cas une manière de récupérer les fonctionnalités offertes par les interfaces de la classe de base.

Effectuer un héritage pour des raisons fonctionnelles est donc le meilleur moyen de récupérer des données inutiles et de rendre le programme plus complexe et plus consommateur de mémoire.

O5. Ne pas faire de code inutilement générique.

Il est inutile de définir des interfaces complexes et des mécanismes génériques pour les communications internes au programme. Ces mécanismes induisent en effet des communications moins directes, et donc plus complexes, moins lisibles et moins performantes, que des mécanismes plus spécifiques et moins génériques.

O6. Définir les grandes entités du programmes et leurs interfaces.

Toutefois, il faut savoir conserver une séparation correcte entre les grandes parties d'un programme pour assurer son évolutivité. Il est donc nécessaire de définir ces grands blocs et définir les interfaces et le niveau de généricité de manière adéquat à chaque niveau de conception. Autrement dit, ce qui est bon au niveau des interfaces entre deux composants de haut niveau ne l'est pas forcément pour des objets de base d'un programme.

O7. Ne pas utiliser d'accesseurs inutiles.

Dans le même ordre d'idées, les accesseurs ne doivent pas être utilisés dans l'implémentation des méthodes de leur classe. En effet, l'utilisation des accesseurs dans l'implémentation ajoute une dépendance inutile envers les interfaces exposées dans l'implémentation, complexifie le code et le rend à la fois moins performant et plus difficile à déboguer en raison des appels de méthodes effectués au lieu des accès directs aux données.

De plus, les accesseurs constituent une partie de l'interface publique de leur classe, et n'ont pas pour but d'être utilisés dans l'implémentation de la classe elle-même. Autrement dit, l'implémentation d'une méthode de classe n'a aucune raison d'utiliser les accesseurs de cette classe, étant donné qu'elle est elle-même spécifique à la structure de données de cette classe.

12.1.4.2. Autres règles d'optimisation

Les règles suivantes sont purement techniques et spécifiques au langage. Elles sont cependant suffisamment simples pour être appliquées en toute circonstances.

O8. Aligner les membres des structures et des classes.

Les données de taille inférieure à la taille des registres du processeur sont généralement alignées sur des adresses multiples de leur taille. Ceci est souvent imposé par le matériel, et même sur les

architectures qui tolèrent des données non-alignées, il est préférable de les aligner pour des raisons de performances.

De ce fait, des zones inutilisées de la mémoire peuvent être insérées par le compilateur entre les différents membres des structures. Par exemple, un caractère et un entier 32 bits seront stockés dans 8 octets consécutifs, les quatre premiers octets étant consommés pour le caractère, qui n'utilise pourtant effectivement qu'un seul d'entre eux.

Ces alignements consomment donc de la mémoire, et du fait qu'ils sont spécifiques à chaque architecture, ils peuvent rendre les structures non portables. Pour éviter cela, tout en conservant des performances optimales, il est recommandé de prendre en compte le problème de l'alignement directement lors de la définition d'une structure. Pour cela, les données peuvent être groupées par blocs de taille identique, ou par types identiques et de taille décroissante.

Par exemple, la structure suivante n'est pas alignée :

```
struct S
{
    char i;     // aligné
    // Trois octets perdus
    int j;      // non aligné !
    char k;     // aligné
    // Un octet perdu
    short l;    // non aligné !
};
```

alors que celle-ci l'est :

```
struct S
{
    int j;
    short l;
    char i;
    char k;
};
```

Moyennant quelques hypothèses sur la taille des types de données, on aurait aussi pu écrire cette structure comme ceci :

```
struct S
{
    char i;
    char k;
    short l;    // aligné : sizeof(short) = 2*sizeof(char)
    int j;      // aligné : sizeof(int) = 2*sizeof(short)
};
```

O9. Utiliser la version préfixe des opérateurs ++ et --.

Dans la plupart des cas, les opérateurs ++ et -- sont utilisés pour incrémenter la valeur d'un objet et manipuler ensuite le résultat. La valeur précédente est donc totalement inutile, et il est donc conseillé d'utiliser les versions préfixes de ces opérateurs.

Les versions suffixes doivent renvoyer la valeur avant modification, et sont donc obligées de conserver

ou de recopier cette valeur avant d'effectuer l'incrément ou le décrément. Cela peut nuire aux performances, car une recopie peut coûter cher et la conservation de la valeur antérieure consomme de la mémoire ou pollue les caches des processeurs. Les versions suffixées devront donc n'être utilisées que lorsque cela ne peut être évité.

```
for (int i=0; i<10; ++i)
{
    // Corps de la boucle.
}
```

12.2. Méthodes et techniques classiques

Nous allons voir dans cette section quelques méthodes et techniques complémentaires qui permettent d'améliorer la qualité des programmes. Quelques grands principes de conception objet seront présentés, ainsi que la manière de réaliser des programmes orientés objets en C. Enfin, les notions d'API et d'ABI seront présentées, suivies de quelques règles permettant de définir des API simples à utiliser.

12.2.1. Méthodologie objet

Les technologies objet sont une avancée indéniable pour les langages impératifs. Elles sont également passionnantes et très intéressantes. Toutefois, elles ne peuvent garantir un succès systématique, et même, mal utilisées, elles peuvent être la cause de programmes extrêmement inefficaces.

C'est pour cela que la programmation orientée objet ne doit pas se réduire à de simples artifices syntaxiques, mais s'orienter dans une méthodologie plus large. Plusieurs « méthodes » ont donc été définies, certaines s'orientant plus sur certains aspects que d'autres. Mais toutes ont pour but d'aider à la conception des programmes, non seulement pour qu'ils soient correctement structurés, mais aussi pour qu'ils répondent aux besoins. Ces méthodes doivent donc s'intégrer dans un processus de développement logiciel plus large que le simple niveau de la programmation, avec lequel elles doivent donc être cohérentes pour être efficaces.

La plupart de ces méthodes sont itératives, et permettent une conception du logiciel par découpage structurel. À chaque itération, la méthode complète est appliquée, d'abord pour les grands composants du logiciel, puis pour les constituants plus petit, et ainsi de suite jusqu'à l'implémentation des briques de base. Généralement, toutefois, une seule itération suffit, et seuls les grands projets nécessitent plus d'une passe.

Il n'est évidemment pas question de décrire ici ces méthodes. Toutefois, les principaux concepts sont toujours utiles et peuvent être présentés afin de fournir un aperçu de ce que ces méthodes préconisent.

12.2.1.1. Définir le besoin et le périmètre des fonctionnalités

L'étape la plus importante dans toute méthode de génie logiciel est assurément l'étape de spécification du besoin. Sans cette étape, il n'est pas possible de savoir ce que l'on doit faire, ni où s'arrêter. Ainsi, si la phase de spécification des besoins et de délimitation des fonctionnalités du programme n'est pas réalisée, dans le meilleur des cas le programme en fera trop et aura coûté trop cher, et dans le pire des cas il aura coûté cher et ne répondra malgré tout pas au besoin. C'est assurément les bogues de ce type qui sont les plus coûteux !

La spécification du besoin fait écho au cahier des charges du maître d'ouvrage et en reprend les termes et les points un à un. En cas d'absence de définition claire du besoin par le maître d'ouvrage, cette phase doit servir à la fois à délimiter le projet et à protéger juridiquement les différentes parties. En effet, en cas de litige, il n'est pas facile du tout de déterminer ce qui est dû et ce qui ne l'est pas. Le maître d'œuvre a donc tout intérêt à effectuer cette formalisation, et au besoin d'aider le maître d'ouvrage à préciser son besoin. Le maître d'ouvrage aussi y a intérêt, puisqu'elle permet de lui donner l'assurance que le logiciel satisfera son besoin.

D'un point de vue plus technique, la définition des besoins et du périmètre est un prérequis pour passer à l'étape suivante. Sans elle, on ne sait a priori ni ce que doit faire le logiciel, ni son cadre d'utilisation. Il ne faut pas oublier qu'au final, l'ordinateur ne devinera pas ce qu'il doit faire, et arrivé au niveau du codage, tout doit être parfaitement spécifié.

Comme il est relativement difficile de formaliser un besoin, les méthodes de conception travaillent souvent en terme d'exemples concrets. Ces exemples sont appelés des cas d'utilisation (« Use Cases » en anglais) dans les langages de modélisation tels que UML (abréviation de « Unified Modeling Language »). Les cas d'utilisation permettent donc de décrire le comportement dans des cas bien précis, et notamment dans le cas d'utilisation nominal, ce qui est fondamental !

Toutefois, une spécification de besoins ne peut se réduire à une liste de diagrammes de cas d'utilisation, car ces diagrammes ne sont ni formels, ni, en général, exhaustifs. Ils ne servent donc qu'à fournir des exemples aisément compréhensibles d'utilisation du produit final, et au minimum à en décrire le cadre d'utilisation nominal et les marches dégradées principales. En ce sens, ils ne sont essentiellement utiles que pour communiquer avec le maître d'ouvrage, et pour l'aider à mieux comprendre ce que fera le logiciel.

C'est pour cela que, en général, les besoins sont ensuite formalisés en terme d'exigences numérotées, et dont la couverture est assurée dans la suite du projet par des matrices de traçabilité. Chaque document de conception devra ensuite référencer ces exigences et y répondre à son niveau, et les plans de tests de validation fonctionnelle devront s'assurer que toutes les exigences sont effectivement couvertes.

12.2.1.2. Identifier les entités et leur nombre

Contrairement aux méthodes de conception fonctionnelles, qui identifient en premier lieu les traitements qui doivent être effectuées, les méthodes objets s'intéressent plus aux entités auxquelles les traitements seront appliqués. Une fois les besoins clairement identifiés, les méthodes objet s'attachent donc à l'identification de ces entités. Ce seront ces entités qui constitueront au final les objets du système.

> **Note :** Bien que l'approche fonctionnelle soit généralement plus facile à appréhender, car en général on sait ce que l'on veut faire, elle ne garantit pas de trouver une structure correcte pour les données du programme, et est donc un facteur de risque très fort d'effets de bord. Inversement, les méthodes de conception objet s'intéressent plus aux entités auxquelles les traitements s'appliquent, garantissant ainsi une cohérence des données forte et une réduction des risques des effets de bord. L'aspect traitement n'est envisagé qu'au final, une fois les entités et leurs interactions connues. Cette approche se justifie par le fait que de toutes manières, les traitements qui doivent être réalisées doivent l'être sur des données, et que les exigences fonctionnelles seront bien traitées, mais dans un contexte structurel sain et correspondant à la réalité du problème.

La phase d'identification n'est en soi pas trop difficile à réaliser. Toutefois, il faut éviter d'entrer dans le détail dès le départ, car tous les objets ne sont pas d'un même niveau fonctionnel. Le processus est

donc itératif, et chaque constituant peut être analysé plus précisément une fois que tous les macro-constituants sont identifiés.

Il est intéressant de déterminer la cardinalité des entités ainsi identifiées. En effet, le nombre de ces entités va influer directement sur les structures de données qui pourront ainsi être choisies afin d'obtenir les meilleures performances pour les principaux cas d'utilisation.

12.2.1.3. Analyser les interactions et la dynamique du système

Les interactions entre les différentes entités peuvent ensuite être décrites. Ces interactions doivent permettre de réaliser les cas d'utilisation et doivent prendre en compte la cardinalité des entités.

Les méthodes utilisent généralement des diagrammes de classes ou de collaboration pour représenter les interactions. Ces diagrammes constituent donc une vue statique des relations entre les différents objets, vue dont les structures de données utilisées lors de l'implémentation seront déduites en phase de conception détaillée.

Il est extrêmement important de décrire également la dynamique du système, c'est-à-dire les messages échangés entre les différents constituants. En effet, c'est elle qui donne du sens aux diagrammes de classes, en indiquant la chronologie des échanges entre ces classes dans les cas d'utilisation. Cela permet également de contrôler que les relations entre les sous-systèmes ont bien été toutes identifiées. Les diagrammes les plus utiles pour cette tâche sont sans doute les diagrammes de séquence.

12.2.1.4. Définir les interfaces

La définition des interfaces doit permettre de réaliser les interactions identifiées précédemment. C'est dans cette phase que les « contrats de service », c'est-à-dire les fonctionnalités qu'ils exposent, sont définis pour les objets. On notera que la description dynamique de l'utilisation des méthodes des interfaces est tout aussi importante que la description de l'interface elle-même, même si souvent l'utilisation peut se déduire implicitement.

Il faut bien garder à l'esprit que les principes objets minimisent les effets de bords et améliorent la réutilisabilité du code, mais que cela se fait au détriment d'un travail de définition et d'implémentation des interface relativement lourd (ce qui est très visible en regardant le moindre programme écrit dans des langages Java ou .Net ou implémentant en C++ des composants COM ou Corba par exemple). Ainsi, il n'est pas rare dans les programmes objets de voir plus de code pour gérer les interfaces que pour gérer la fonctionnalité elle-même.

Il est donc important, afin de ne pas trop souffrir de ce revers de médaille, de définir les interfaces des classes dans le but de permettre les interactions de la manière la plus simple et la plus claire possible. Les deux axes décrits ci-dessous peuvent être suivis pour cela.

Premièrement, il faut savoir exploiter la généricité à bon escient. La généricité est une bonne chose, quand elle est justifiée, mais elle peut devenir néfaste dans le cas contraire.

Il est recommandé de définir des interfaces communes à des objets de classe semblable, afin de permettre leur manipulation de manière polymorphique. Cependant, cela ne doit pas être un but en soi, et il ne faut pas s'acharner à faire rentrer des carrés dans des ronds. Par conséquent, les interfaces doivent être définies en fonction de la réalité du problème, et si les classes d'objets manipulés ne sont pas semblables, et bien tant pis, c'est que de toutes manières un traitement spécifique doit leur être appliqué.

Deuxièmement, il faut chercher à minimiser les échanges entre objets. Il est important de s'assurer que les interfaces permettent d'effectuer les opérations le plus efficacement et le plus simplement possible.

Pour cela, les informations nécessaires au traitements doivent être communiquées aux méthodes qui en ont besoin, éventuellement par l'intermédiaire d'une référence d'objet.

Cette règle est particulièrement importante dans le cas où plusieurs données doivent être récupérées sur un objet partagé avec plusieurs autres objets, ou lorsqu'un objet doit signaler un événement à un autre objet. Ainsi, il est préférable de définir des méthodes permettant de récupérer un ensemble de valeurs en une fois qu'une multitude d'accesseurs (toujours très lourds à écrire et à implémenter). De même, l'ensemble des paramètres relatifs à un événement doit être fourni lors de la notification de cet événement.

Les raisons de cette dernière règle sont multiples. En fait, ne pas fournir les informations relatives à un événement complexifie le traitement des objets serveurs et n'est pas fiable dans un système complexe :

- Cela impose de conserver les informations relatives à l'entité qui a généré l'événement au niveau de l'objet serveur. Cela peut être artificiel, et la question de la durée de la conservation de ces informations se pose immédiatement.
- Lorsque la notification signale un changement d'état, il n'est pas possible de garantir à un client que l'état qu'il obtiendra en appelant un accesseur sera celui qui a provoqué la notification de changement, surtout dans un contexte multithreadé.
- Dans certaines circonstances, comme dans le cadre d'une communication réseau par exemple, un appel supplémentaire peut être plus coûteux que le transfert de toutes les informations lors de la notification.
- Enfin, la multiplication des appels tend à rendre le programme plus complexe, plus difficile à suivre, et plus risqué dans un environnement multithreadé.

> **Note :** Il arrive quelquefois que l'on ne soit intéressé que par le fait qu'un événement se produise, et pas par le détail de cet événement. Toutefois, on ne peut pas, de manière générale, préjuger de l'usage qui pourra être fait d'un événement fourni par un objet. Par conséquent, il faut prévoir de fournir toutes les informations, quitte à perdre un peu de temps pour le cas où elles seraient ignorées par le client.

12.2.1.5. Réalisation et codage

Si les étapes précédentes sont correctement réalisées, la phase de réalisation doit pouvoir se faire avec une bonne visibilité.

Les seuls risques à ce niveau sont les risques liés à l'environnement, aux outils ou aux technologies employées. En général, il n'est jamais inutile de procéder à des tests simples sur les points clés pour valider une technologie que l'on ne connaît pas parfaitement. Cela peut éviter bien de mauvaises surprises. De manière générale, une bonne technologie est une technologie appropriée au problème à résoudre et à son environnement.

12.2.1.6. Les tests

Les tests consistent à vérifier que les spécifications sont bien implémentées. Cela implique naturellement que sans spécifications, on ne peut pas faire de tests exhaustifs. Tout au plus peut-on tester les cas nominaux et les cas d'erreurs triviaux, au gré de l'imagination de celui qui les réalise.

Il est possible de définir des tests pour chacune des phases de conception. Les tests unitaires sont les tests qui se placent au niveau du codage, ils ont pour but de vérifier que les objets se comportent correctement et sont utilisables via leurs interfaces. La réalisation de ces tests peut nécessiter le développement d'outils hôte pour les objets testés. Il est évident qu'il est plus facile de tester unitairement des objets autonomes que des objets qui nécessitent un environnement complexe ou dont l'encapsulation est imparfaite.

Les tests d'intégration se situent au niveau conception et définition des interfaces. Ils ont pour but de vérifier que les différentes entités identifiées en conception parviennent à communiquer et à s'intégrer les uns avec les autres correctement. Ce sont assurément les tests les plus longs, puisqu'ils reviennent à faire fonctionner le système.

Enfin, les tests de validation sont les plus importants, puisqu'ils ont pour but de vérifier que les exigences des spécifications de besoin sont bien couvertes. Un logiciel qui ne vérifie pas ces tests n'est pas forcément inutilisable, mais pas pour faire ce pour quoi il a été developpé.

12.2.1.7. Les design patterns

En général, la plupart des problèmes rencontrés pendant la conception sont classiques et ont déjà été résolus maintes fois, dans de multiples circonstances. En fait, les problèmes de conception sont presque toujours les mêmes, et il est possible de les classer en quelques grandes catégories.

Cela implique que, bien souvent, les solutions qui ont été adoptées dans certaines circonstances sont utilisables ou peuvent être adaptées facilement pour résoudre le même problème dans d'autres circonstances. Ces problèmes de conception ont donc souvent déjà été résolus par d'autres programmeurs, et les solutions trouvées vérifiées comme fonctionnant et apportant une réponse appropriée.

De ce fait, il n'est en général pas nécessaire de réinventer la roue, au risque de faire une erreur de conception ou un mauvais choix que d'autres programmeurs ont déjà su éviter (ou faite !). Ainsi, il suffit de trouver « la » solution correspondante au problème à résoudre, et de l'appliquer directement. C'est dans cet esprit que les « design patterns » ont été inventés.

Un « design pattern » (motif ou élément de conception de base) n'est rien d'autre qu'un modèle de solution générique reconnu et applicable à un type de problème donné. Les design patterns ne couvrent évidemment pas toutes les situations, mais ils peuvent aider de diverses manières :

- ils diminuent la charge du programmeur car ils apportent une réponse toute faite ;
- ils réduisent les risques de mauvaise conception, car ils assurent que cette réponse est correcte ;
- ils sont facilement identifiables, et sont donc facilement reconnaissable dans un modèle objet ou une application, permettant ainsi une compréhension rapide du modèle objet ;
- ils assurent une certaine cohérence de conception entre différents projets, permettant potentiellement une meilleure intégration ou interchangeabilité.

On aura donc tout intérêt à utiliser les design patterns. Pour cela, il suffit simplement d'en connaître les principaux, et de faire l'effort d'identifier les problèmes que l'on peut résoudre grâce à eux en phase de conception.

> **Note :** Bien que les design patterns fournissent une solution générique à un grand nombre de problèmes, ils ne garantissent pas que leur implémentation sera correcte. Ainsi, un soin particulier doit être apporté lors de leur implémentation, et les considérations classiques de codage devront toujours être prises en compte.

12.2.2. Programmation objet en C

Nombre de projets choisissent encore le langage C sans pour autant vouloir renoncer aux avantages des techniques objets. La raison de ce choix est généralement de garantir la légèreté du programme ou de pouvoir s'abstraire des spécificités des compilateurs C++ (notamment en ce qui concerne le nommage des symboles) et réaliser ainsi des bibliothèques extrêmement portables ou devant être utilisées avec d'autres langages de programmation.

Heureusement, il existe une technique élégante permettant de programmer des objets en C. Cette technique reproduit les constructions que les compilateurs C++ utilisent généralement en interne pour gérer les classes et les fonctions virtuelles. Comme elle se rencontre couramment, il n'est pas inutile de la décrire brièvement.

12.2.2.1. Principe de base

Le principe de base est de stocker l'ensemble des informations des objets dans une structure, et de fournir systématiquement cette structure en paramètre aux méthodes qui permettent de la manipuler. Cela permet de reproduire la notion d'objet simple et de méthodes, le premier paramètre faisant office de pointeur `this` pour les méthodes de la classe. Tous les objets de même type étant des instances d'une même structure, les méthodes sont capables de travailler avec différentes instances simplement en passant les références sur ces instances en paramètre à chaque appel :

```
stringlist_add(liste, "Une chaîne");
stringlist_add(liste, "Une autre chaîne...");
```

Cette technique reprend bien les principes de base en réduisant la portée des variables manipulées à la structure de l'objet, tout en évitant de fournir un nombre gigantesque de paramètres locaux.

Les objets passés aux méthodes qui les manipulent doivent être définis de la manière la plus opaque possible. Ainsi, ils ne peuvent être modifiés que par les fonctions dédiées à cet effet. Cela permet bien sûr de mieux contrôler la manière dont les données du programme ou de la bibliothèque sont manipulées par les programmes clients.

La manière la plus simple de réaliser en C un type de données opaque est d'utiliser des pointeurs sur des structures non définies. Cela se fait simplement comme suit :

```
struct _ma_structure;
typedef _ma_structure *ma_structure_t;
```

Les méthodes de manipulation n'ont ensuite à utiliser plus que le type de données ma_structure_t. La structure struct _ma_structure n'a pas à être connue des programmes clients (elle est juste déclarée), et peut n'être définie que dans les fichiers d'implémentation des méthodes de l'interface de l'objet.

12.2.2.2. Définition d'interfaces

Il est possible de pousser plus loin la technique précédente, en définissant des interfaces utilisables sur les objets. Pour cela, il suffit d'associer les méthodes de ces interfaces à chaque objet auxquelles elles peuvent être appliquées. Pratiquement, les interfaces sont définies sous la forme de tableaux de pointeurs de fonctions contenant les adresses des méthodes de l'interface. Ces tableaux sont ensuite référencés dans les objets qui implémentent ces interfaces. Si la référence à l'interface est stockée à la

même position pour plusieurs objets (par exemple en première position...), alors ceux-ci peuvent être manipulés polymorphiquement.

Par exemple, on peut définir une interface pour des objets « nommables » de la manière suivante :

Exemple 12-8. Définition d'interface en C

```
/* Déclaration du type de base des objets nommés */
struct _namedobject;
typedef struct _namedobject *namedobject_t;

/* Prototypes des méthodes des objets nommés : */
typedef void (*destroy_namedobject)(namedobject_t);
typedef void (*set_namedobject_name)(namedobject_t, const char *szName);
typedef void (*print_namedobject_name)(namedobject_t);

/* Structure de l'interface des objets nommés : */
typedef struct
{
    /* Méthodes applicables : */
    destroy_namedobject destroy;
    set_namedobject_name set_name;
    print_namedobject_name print_name;
} fnamedobject_t;

struct _namedobject
{
    /* Table des méthodes de l'interface : */
    fnamedobject_t *ops;
};
```

Si l'on veut implémenter cette interface pour une classe d'objet, on peut simplement déclarer des fonctions dont la signature est identique à celle de l'interface :

```
/* Déclaration du type des objets : */
struct _object1;
typedef struct _object1 *object1_t;

/* Déclaration du constructeur : */
extern object1_t create_object1(const char *szName);

/* Méthodes applicables : */
extern void destroy_object1(object1_t object);
extern void set_object1_name(object1_t object, const char *szName);
extern void print_object1_name(object1_t object);
```

L'implémentation de la classe se fait alors de la manière suivante :

Exemple 12-9. Implémentation d'un objet en C

```
/* Implémentation de la classe object1. */

/* Définition de la classe object1 : */
struct _object1
{
    /* Implémentation de l'interface des objets nommés : */
    struct _namedobject m_base;
    /* Structure de données de la classe : */
    char *m_szName;
};

/* Table des méthodes de la classe object1 : */
static fnamedobject_t object1_methods =
{
    (destroy_namedobject) &destroy_object1,
    (set_namedobject_name) &set_object1_name,
    (print_namedobject_name) &print_object1_name
};

/* Implémentation des méthodes de la classe objet1 : */
object1_t create_object1(const char *szName)
{
    object1_t obj = (object1_t) malloc(sizeof(struct _object1));
    if (obj != NULL)
    {
        // Initialisation :
        obj->m_base.ops = &object1_methods;
        obj->m_szName = NULL;
        // Appel de la méthode set_name :
        set_object1_name(obj, szName);
    }
    return obj;
}

void destroy_object1(object1_t object)
{
    free(object->m_szName);
    object->m_szName = NULL;
    free(object);
}

void set_object1_name(object1_t object, const char *szName)
{
    char *szNewName = NULL;
    int len = strlen(szName);
    szNewName = (char *) malloc(len + 1);
    if (szNewName != NULL)
    {
        strcpy(szNewName, szName);
        if (object->m_szName != NULL)
        free(object->m_szName);
        object->m_szName = szNewName;
    }
}
```

```
void print_object1_name(object1_t object)
{
    if (object->m_szName != NULL)
    {
        printf("%s\n", object->m_szName);
    }
    else
    {
        printf("objet anonyme\n");
    }
}
```

On notera que le tableau de pointeurs des méthodes de la classe est commun à tous les objets de cette classe. Plusieurs structures identiques peuvent partager des méthodes communes, mais en général, chaque structure a ses propres fonctions, ce qui correspond au mécanisme des fonctions virtuelles du C++. En particulier, la fonction `destroy` se charge de détruire correctement les objets de la classe à laquelle elle appartient, faisant ainsi l'équivalence avec les destructeurs virtuels du C++.

La manipulation des objets nommables est alors simple et peut se faire, moyennant un simple transtypage, de manière identique pour tous les objets qui implémentent l'interface :

Exemple 12-10. Utilisation d'un objet en C

```
#include <stdlib.h>
#include "object1.h"

int main(void)
{
    /* Création d'un objet nommé : */
    object1_t o = create_object1("Un objet en C...");
    /* Utilisation directe de cet objet nommé : */
    print_object1_name(o);
    /* Utilisation de l'objet via son interface : */
    namedobject_t named = (namedobject_t) o;
    named->ops->set_name(named, "Hello World !");
    named->ops->print_name(named);
    named->ops->destroy(named);
    return EXIT_SUCCESS;
}
```

12.2.2.3. Compatibilité binaire

Outre le fait que la structure des données n'est pas exposée dans les fichiers d'en-tête, la technique qui vient d'être présentée permet de conserver la compatibilité source et binaire des programmes si l'implémentation change, et permet d'accroître les performances en simplifiant les passages de paramètres inutiles dans les fonctions. En effet, le seul code qui manipule la structure des objets se trouve dans les fichiers d'implémentation de l'objet lui-même. Les programmes clients n'utilisent que des pointeurs sur ces objets, et ne font aucune hypothèse sur leur structure.

Notez toutefois que la conservation de la compatibilité binaire impose de ne pas modifier les interfaces tant au niveau de l'ordre des méthodes des objets que du nombre et de la sémantique de leurs

paramètres. Les codes de retours ne peuvent généralement pas être modifiés non plus, pas même étendus. En revanche, il est faisable de changer le nom des méthodes (incompatibilité au niveau source uniquement) ou d'en rajouter (extension d'interfaces).

12.2.3. ABI et API

Une « API » (abréviation de « Application Programming Interface ») est la spécification descriptive d'une interface d'un programme ou d'une bibliothèque au niveau langage. Généralement, les API sont définies comme un jeu de fonctions et des différentes valeurs possibles pour leurs paramètres. Une « ABI » (abréviation de « Application Binary Interface » est la spécification d'une interface bas niveau. L'ABI s'intéresse donc plus particulièrement à la manière dont les fonctions de l'interface sont appelées au niveau binaire.

Le choix d'une ABI et la définition de l'API d'une bibliothèque sont des choix importants, qui sont conditionnés par l'usage qui sera fait de cette bibliothèque et par les possibilités d'évolution qu'elle doit avoir.

12.2.3.1. Choix de l'ABI

Les ABI définissent l'ensemble des règles qui permettent de savoir comment les fonctionnalités d'une bibliothèque sont utilisées au niveau binaire, sur une plateforme donnée. Elles décrivent un certain nombre d'aspects, parmi lesquels on retrouve les conventions d'appel et les noms utilisés dans les fichiers objets pour l'édition de liens.

Les conventions d'appel doivent être parfaitement spécifiées lors de la définition d'une ABI. Les conventions d'appel spécifient en particulier la manière dont les paramètres sont passés à une fonction, ainsi que qui, de l'appelant ou de l'appelé, doit détruire les paramètres.

En général, la convention la plus utilisée est la convention du langage C (les paramètres sont tous passés par la pile, dans l'ordre inverse de passage dans l'écriture de l'appel de fonction, et sont libérés par l'appelant). Il s'agit également de la convention utilisée par défaut pour le C++ et pour de nombreux autres langages. Toutefois, il existe d'autres conventions, qui sont moins courantes et moins portables.

Généralement, les conventions d'appel sont spécifiées à l'aide d'un attribut. La manière de fixer les attributs sur les fonctions et les variables globales n'est pas spécifiée et encore moins portable. Elle se fait souvent avec un mot-clé spécifique, que l'on place avant l'identificateur ainsi qualifié :

```
int __cdecl i;                          /* Spécification de convention d'appel C en Visua
void __attribute__((stdcall)) f(int);  /* Spécification de convention d'appel
                                          STDCALL de Windows avec GCC */
```

L'ABI spécifie également les noms utilisés dans les fichiers objets pour représenter les identificateurs. Ces noms sont utilisés pour l'édition de liens, afin de retrouver les emplacements mémoire utilisés par les variables et le code binaire des fonctions.

Si les noms des identificateurs C sont standardisés et peuvent être facilement connus à partir de leurs déclarations (il suffit de préfixer le nom de l'identificateur par un '_'), il n'en va pas de même pour les identificateurs C++. En effet, les compilateurs C++ ajoutent des informations de typage aux noms des identificateurs du programme dans les fichiers objets, afin de permettre les surcharges de fonctions et les contrôles de type. On dit que ces noms sont « décorés ». De ce fait, les noms des fonctions

exportées par les bibliothèques C++ sont rarement utilisables avec un autre compilateur que celui qui les a générés, et encore moins à partir d'un autre langage.

La situation est en voie d'amélioration, puisqu'une norme précisant l'ABI C++, et donc en particulier le format des noms décorés, a été diffusée et semble admise par la plupart des éditeurs de compilateurs. Toutefois, les implémentations de cette ABI ne sont pas encore totalement stabilisées et, surtout, les compilateurs sous Windows ne la respectent pas. Microsoft ne semble manifestement pas vouloir modifier Visual C++ pour cela, n'ayant pas besoin ou intérêt à assurer une intéropérabilité avec les éditeurs de compilateurs pour les autres systèmes. Enfin, même en supposant que tous les compilateurs C++ utilisent les mêmes décorations pour les noms d'identificateurs dans les fichiers objets, ces noms ne seraient toujours pas facilement utilisables à partir d'autre langages que le C++.

Il est donc important, lorsqu'on réalise une bibliothèque ou des fonctions utilitaires, de déterminer les conditions d'utilisation de cette bibliothèque. Le choix du langage C++ restreint fortement son champ d'application et l'interopérabilité. Si cette bibliothèque doit être utilisée à partir d'autres langages, il est préférable d'utiliser la directive d'édition de liens « `extern "C"` » pour exposer les noms des identificateurs publics avec les conventions du C, qui sont reconnues universellement. Cela a également pour avantage de réduire le risque d'exposition des particularismes du langage C++ inaccessibles à d'autres langages, comme par exemple les surcharges des fonctions ou les exceptions.

12.2.3.2. Définition de l'API

La définition d'une API pour un programme est un exercice difficile, car une fois définie, elle ne peut être changée sans provoquer de rupture de la compatibilité au niveau du code source, et même souvent au niveau du code compilé. De plus, une mauvaise API peut rendre la programmation lourde et complexe, voire inefficace et risquée. Cette section se propose donc de souligner quelques points importants dans la définition d'une API.

12.2.3.2.1. Simplicité d'utilisation

Réaliser une API simple d'emploi n'est manifestement pas une chose facile. Quelques règles peuvent cependant aider le programmeur dans cette tâche.

Avant tout, il faut que les fonctions de l'API soient faciles à appeler, et donc qu'elles aient un nombre réduit de paramètres. Pour cela, les techniques objet peuvent grandement aider, puisque toutes les données membres d'un objet sont fournies implicitement en paramètre via le pointeur sur l'objet. Bien entendu, ces données membres doivent généralement être initialisées lors de la création d'un objet, mais cela n'est fait qu'une seule fois.

> **Note :** Certaines API utilisent une notion de contexte global pour réduire le nombre des paramètres (par exemple, matrice de transformation dans OpenGL et masque des droits des fichiers à leur création dans Posix). Toutefois, cela n'est pas à recommander, car ce contexte peut être modifié de manière implicite d'une part, et ne permet pas d'utiliser l'API avec plusieurs clients d'autre part. Si un contexte doit être utilisé pour simplifier les appels de fonctions, alors autant que ce contexte soit contenu dans un objet et que les fonctions soient des méthodes de cet objet.

Un autre aspect important pour la simplicité d'utilisation d'une API est la possibilité d'utiliser des entités différentes de manière uniforme. Là encore, les notions objets sont utiles, car il est possible de manipuler plusieurs entités semblables par polymorphisme. Par exemple, il est possible de faire une synchronisation avec l'ensemble des objets systèmes du noyau de Windows. Ces objets sont tous représentés par un type d'objets unique du système, et peuvent être manipulés de manière uniforme

via un jeu de fonctions travaillant sur ce type. De même, la manipulation des fichiers, des disques et des périphériques au sens général sous Unix se fait de manière uniforme via la notion de fichier et les fonctions de manipulation des fichiers (bien que les interfaces réseau fassent exception à la règle).

> **Note :** On veillera à ne pas faire les erreurs décrites précédemment dans la définition d'une API objet. Par exemple, un abus de généricité peut être très néfaste. En effet, cela peut entraîner une profusion de paramètres dans les méthodes pour en spécialiser le comportement lorsqu'elles sont appliquées à des objets qui ne se prêtent pas au modèle des interfaces utilisées.

12.2.3.2.2. Interfaces synchrones et asynchrones

Généralement, les appels de méthodes sont bloquants et ne se terminent qu'une fois le traitement de la méthode terminé. Cependant, il arrive des situations où ce traitement peut être long, et l'appelant peut ne pas vouloir en attendre la fin pour exécuter un autre traitement simultanément. Il est possible dans ce cas de faire en sorte que la méthode appelée se contente d'amorcer le traitement et rende la main immédiatement, le composant cible effectuant alors le traitement en parallèle grâce à un autre processus, un autre thread ou toute autre technique appropriée d'exécution concurrente. La méthode appelée est alors dite asynchrone, et le résultat du traitement (données fournies en retour ou compte-rendu d'exécution) doit être récupéré ultérieurement.

Il existe plusieurs techniques pour récupérer les informations relatives à un traitement asynchrone :

- soit la méthode appelée fournit en retour un objet réponse grâce auquel on peut attendre la fin de l'opération, et éventuellement récupérer les résultats ;
- soit l'appelant fournit une fonction de rappel qui sera appelée par le composant qui effectue le traitement une fois celui-ci fini ;
- soit un message est envoyé à l'appelant par le composant qui effectue le traitement, via un canal de communication standard.

Les API synchrones sont souvent les plus simples à utiliser. Toutefois, ce ne sont pas les plus performantes. Comme il est toujours possible de simuler une API synchrone facilement à partir d'une API asynchrone en attendant explicitement la fin de son exécution, il est recommandé de proposer les deux modèles. Ainsi, chaque programme pourra utiliser la version des méthodes la plus appropriée à ses besoins.

12.2.3.2.3. Gestion des allocations mémoire

Il existe de nombreuses méthodes de gestion de la mémoire, et les blocs de données alloués par un allocateur donné ne peuvent généralement pas être libérés par un autre allocateur. Cela implique que les programmes clients d'une API ne peuvent généralement pas utiliser leur propre allocateur pour libérer les ressources qu'ils ont obtenues via cette API. Ce point est encore plus important pour les bibliothèques utilitaires destinées à être utilisées dans un programme écrit dans un autre langage que le C/C++.

Le fait de documenter la manière dont les ressources sont allouées ne suffit généralement pas pour supprimer ce problème. En effet, sur certaines plateformes, l'allocateur mémoire utilisé dépend de la configuration du projet (Unicode ou non, Debug ou Release, édition de liens statique ou dynamique avec la bibliothèque C, etc.). Ainsi, même en sachant comment il faudrait libérer une ressource, le programme client ne peut le réaliser, car il n'a pas accès à l'allocateur qui l'a allouée.

Il est donc recommandé, afin de s'abstraire des problèmes de conflits entre allocateurs mémoire, de toujours fournir les fonctions qui permettent de libérer les ressources que le programme peut obtenir via l'API. Ainsi, la libération des ressources se fait dans le contexte où elles ont été allouées, par le code qui les a allouées. Cela peut complexifier légèrement la gestion des ressources dans les programmes clients, surtout s'ils utilisent plusieurs bibliothèques utilitaires simultanément. Toutefois, l'utilisation des allocateurs mémoire des bibliothèques peut être rendue très naturelle si la conception de l'API est objet et que tous les objets fournissent une méthode de destruction.

12.2.3.2.4. Allocation des valeurs de retour

Lorsqu'une fonction doit renvoyer un résultat de taille non déterminable lors de la définition de l'API, il est nécessaire d'utiliser une allocation dynamique de mémoire. Si cela ne pose pas problème particulier dans un programme classique, cela peut être gênant dans le cadre de la définition d'une API.

Classiquement, deux approches sont possibles pour gérer ce genre de situation. La première est de considérer que les résultats retournés par la fonction appelée sont stockés dans une zone mémoire allouée par cette dernière. Cette zone doit donc être libérée par le code appelant, de manière compatibles à la méthode d'allocation utilisée. La fonction de libération de mémoire à utiliser est dans ce cas, comme on l'a vu dans la section précédente, la fonction fournie par l'API à cet effet.

Par exemple, la fonction `strdup`, disponible sur les systèmes Posix et Unix98, effectue une copie d'une chaîne de caractères et retourne cette copie. La mémoire utilisée pour stockée cette copie est allouée avec la fonction `malloc` et doit donc être libérée avec la fonction `free` :

```
// Libération de la mémoire allouée par strdup :
free(szCopy);
```

Cette technique convient généralement dans de nombreux cas. Toutefois, elle impose un allocateur mémoire spécifique (en l'occurrence, celui de la bibliothèque C dans notre exemple). De ce fait, ce bloc mémoire ne peut être manipulé comme un bloc alloué par le programme lui-même, ce qui impose de recopier les données si l'on veut conserver une indépendance du programme vis à vis des bibliothèques qu'il utilise, ou tout simplement une certaine cohérence.

Par exemple, si un programme reçoit une chaîne de caractères allouée dynamiquement par une bibliothèque « mylib », et doit les libérer avec la fonction « `mylib_free` », il ne peut se permettre de stocker cette chaîne directement, au risque d'appeler `free` ou `delete[]` sur le pointeur de la chaîne et de provoquer ainsi une erreur mémoire. Par conséquent, il doit en faire une copie dans un bloc alloué avec son propre allocateur. Cela peut affecter les performances mais, comme on l'a dit, reste impératif dans le contexte d'un grand programme pour conserver la cohérence du programme et éviter les bogues.

La deuxième approche consiste à imposer à l'appelant de fournir une zone de mémoire de taille suffisante pour y stocker les résultats. Dans ce cas, l'appelant est maître de l'allocateur utilisé, et n'a donc pas à recopier les données retournées par la fonction appelée.

Toutefois, la difficulté est ici de contrôler que la taille de la zone mémoire est suffisante pour y stocker les données. Cette taille doit donc toujours être fournie en paramètre à la fonction appelée, et un code d'erreur spécifique signalant que la taille est insuffisante doit être défini. Le problème se décale alors vers celui de la détermination de la taille nécessaire pour stocker les données avant l'appel de la fonction. Généralement, la fonction qui retourne le résultat peut être appelée avec un pointeur nul sur le bloc de données, et elle renvoie dans ce cas la taille nécessaire pour y stocker les résultats. Malheureusement, cette solution peut être lente (dans le cas où la détermination de la taille exige un

traitement conséquent, parfois même l'exécution de l'opération « à blanc ») et non suffisante (si la taille peut varier d'un appel à un autre).

Par exemple, la fonction GetComputerName de Windows permet de retourner le nom de l'ordinateur dans un tampon alloué par l'appelant. Elle prend en paramètre le pointeur sur ce tampon et un pointeur sur un entier contenant la longueur de ce tampon. Si cette longueur est insuffisante, cet entier est modifié pour recevoir la taille nécessaire pour que l'appel réussisse complètement, et un code d'erreur est retourné.

```
char *szBuffer = NULL;
// Récupération de la taille nécessaire à l'appel :
unsigned long ulSize = 0;
GetComputerName(szBuffer, &ulSize);
// Allocation de la mémoire du tampon :
ulSize++; // Pour le nul terminal
szBuffer = new char[ulSize];
// Appel de la fonction :
GetComputerName(szBuffer, &ulSize);
```

Ce simple exemple suffit pour démontrer que cette technique est nettement plus lourde, car elle impose d'appeler plusieurs fois de suite la fonction. De plus, il faut généralement contrôler à chaque appel si toutes les données ont bien été reçues, dans l'éventualité où les données à récupérer changent de taille entre chaque appel. Ainsi, le code de l'exemple précédent ne fonctionne pas si le nom de l'ordinateur est changé entre le premier et le deuxième appel à `GetComputerName`. En revanche, dans les situations où les données peuvent être récupérées par blocs (par exemple pour des opérations de type entrée/sortie), cette solution est extrêmement pratique puisqu'elle permet de récupérer les données directement dans la zone mémoire spécifiée par l'appelant, y compris dans les couches systèmes des périphériques d'entrée / sortie.

Comme on peut le constater, chaque technique a ses avantages et ses inconvénients. Si l'on peut garantir que l'allocateur mémoire est unique dans tous le programme (par exemple l'allocateur de la bibliothèque C utilisée sous forme de bibliothèque de liaison dynamique), les copies de données imposées par la première technique ne sont plus nécessaires et le choix est vite fait. Dans les mécanismes d'entrée / sortie, la deuxième solution est généralement plus classiquement adoptée car elle s'avère la plus performante globalement.

> **Note :** En aucun cas une fonction de bibliothèque ne doit retourner de données statiques. Cela est une porte ouverte aux effets de bords et n'est pas utilisable dans un contexte multithreadé (un thread peut en effet modifier le résultat obtenu par un autre de manière imprévisible dans ce cas).

12.3. Considérations système

Certaines considérations système doivent être prises en compte lors de la réalisation des programmes. Elles peuvent en effet influer fortement sur sa portabilité ou sa conception. Les sections suivantes s'attachent donc à décrire les plus importantes d'entre elles.

12.3.1. La sécurité

La sécurité doit être prise en compte dès le début de la conception d'un programme, parce que les règles de sécurité sont de plus en plus appliquées au pied de la lettre d'une part, et parce que c'est une notion transverse d'autre part.

Les politiques de sécurité correctes impliquent de définir plusieurs remparts, afin de s'assurer que si une faille est exploitée l'attaquant n'obtiendra pas le contrôle total du système. Cela implique que les opérations qui traversent plusieurs couches systèmes sont susceptibles de se conformer aux politiques de sécurité de l'ensemble de ces couches. Cela donne autant de raisons à une application de ne pas fonctionner, et de rendre son déploiement plus difficile.

Inversement, une application constituée de différents composants est aussi sûre que le plus faible d'entre eux. Autrement dit, un rempart n'est pas plus fort que le plus faible de ses murs ou la plus faible de ses portes. Cela implique encore une fois que les applications qui ont pour vocation de s'intégrer dans un système sûr doivent prendre en compte la sécurité à tous les niveaux, faute de quoi il y aura sans doute une faille dans un de ses composants.

Or les mécanismes de sécurité sont souvent intrusifs. Dans le cas le plus strict, chaque appel système peut être soumis à autorisation, et donc est susceptible d'échouer si l'application s'exécute dans un contexte de sécurité non privilégié. Cela signifie que les erreurs doivent être systématiquement traitées par l'application.

Bien entendu, une application correctement écrite traite déjà les erreurs et adopte un comportement adéquat. Cependant, les erreurs provenant d'un accès refusé à une fonction ne sont pas naturellement prises en compte par les programmeurs, car il est difficile d'imaginer l'ensemble des opérations qui peuvent échouer pour des raisons de sécurité. De plus, ce type d'erreur relève de l'erreur de configuration ou du contexte, et non d'un problème fonctionnel. La manière de réagir à ces erreurs peut donc être différente de celle dont la cause est, par exemple, un manque de ressources.

En particulier, l'aide au diagnostic est toujours un plus pour l'utilisateur. En effet, les erreurs de programmes dues à un manque de droits sont très mal perçues des utilisateurs, qui se sentent fatalement frustrés, d'autant plus que ce qu'ils recherchent, c'est avant tout que le programme fonctionne. La politique appliquée est donc souvent de désactiver totalement la sécurité, pour le pire et au détriment de tout le monde ! Par ailleurs, ces erreurs sont la cause d'une perte de temps considérable lors des phases de déploiement. Par conséquent, si l'application est capable d'indiquer la raison de son dysfonctionnement avec précision, l'utilisateur sera plus enclin à l'installer conformément aux règles de l'art et à la faire fonctionner correctement, et les diagnostics seront plus aisés lors du déploiement.

Le choix est donc simple : soit la sécurité est prise en compte dès le début des spécifications du logiciel, soit elle est ignorée. Dans ce cas, une provision importante doit être faite pour l'aléa induit par une sécurité mal maîtrisée, et le risque des problèmes qui en découleront couvert en conséquence.

12.3.2. Le multithreading

Le multithreading est une fonctionnalité extrêmement puissante et permettant d'obtenir des programmes performants et réactifs. Cependant, comme nous allons le voir, il induit des contraintes supplémentaires, et requiert de le prendre en compte au niveau de la conception des programmes.

12.3.2.1. Généralités

Le multithreading est la possibilité qu'ont les programmes de disposer de plusieurs flux d'exécution, appelés classiquement « threads », s'exécutant de manière concurrentielle ou parallèle.

Généralement, les systèmes d'exploitation sont capables d'exécuter les threads de manière concurrente, en affectant à chacun d'eux les ressources de calcul de la machine à tour de rôle. Chaque thread s'exécute donc un court laps de temps appelé « time slice » (littéralement, « tranche de temps »), puis redonne la main au système, qui peut basculer vers un autre thread (on appelle cette opération un « changement de contexte »). Ainsi, même si une seule unité de calcul est présente dans la machine, tout apparaît comme si les threads s'exécutaient en parallèle.

La manière dont le basculement d'un thread à un autre se fait est une caractéristique du système d'exploitation. Si les threads doivent rendre la main volontairement pour provoquer un changement de contexte, il s'agit de multitâche coopératif. Si, en revanche, le système s'occupe de tout et suspend les threads qui ont épuisé leur quota de ressource de calcul pour exécuter d'autres threads, il s'agit de multitâche préemptif. Dans ce cas, les ressources de calcul sont attribuées en fonction de priorités, et les préemptions se font à échéance du « time slice » ou lorsqu'un thread de priorité supérieur devient éligible.

En pratique, le multithreading est utilisé pour rendre indépendant les temps de réaction d'une partie d'un programme vis à vis d'autres parties du même programme. Ainsi, le programme peut effectuer les opérations longues ou susceptibles de bloquer lors de l'accès à des ressources lentes dans des flux d'exécution dédiés, et continuer à fonctionner normalement en attendant que ces opérations se terminent.

Par exemple, un programme peut copier des fichiers, récupérer des données sur le réseau, ou effectuer un calcul long tout en continuant à interagir avec l'utilisateur. Tout cela est réalisable sans multithreading, en utilisant des mécanismes d'entrée/sortie asynchrones ou en découpant les calculs en morceaux et en les exécutant quand l'utilisateur ne réagit pas, mais cela complique sérieusement la programmation. En effet, les tâches sont dans ce cas destructurées et requièrent de maintenir un état courant pour chaque opération en cours d'exécution. Le multithreading parvient au même résultat, mais en laissant au système le soin de découper, si nécessaire, les flux d'exécution des différents threads.

Le multithreading est également un moyen efficace d'augmenter les performances. En effet, bloquer un programme en attendant une ressource lente n'est généralement pas efficace, puisque pendant ce temps l'unité de calcul ne travaille pas. De même, il est préférable, quitte à attendre, d'attendre plusieurs ressources lentes simultanément (par exemple une écriture sur disque et la réception de données provenant du réseau). Il est donc recommandé, en général, d'utiliser un thread pour communiquer avec chaque entité avec lequel le programme doit travailler. Bien entendu, les mécanismes d'entrée/sortie asynchrones sont là aussi utilisables, mais simplement moins pratiques.

Enfin, le multithreading et le multitâche sont les seuls moyens mis à la disposition des programmes pour bénéficier des ressources de calcul multiples des machines multiprocesseurs ou hyperthreadées (machines dont le processeur simule plusieurs processeurs logiques pour optimiser le taux d'utilisation de l'unité de calcul par rapport aux unités d'accès à la mémoire et de décodage des instructions). En effet, sur ces machines, le système d'exploitation peut répartir les threads sur les unités de calcul et les faire fonctionner réellement en parallèle. Ainsi, si les threads sont totalement indépendants, le gain théorique peut aller jusqu'à un facteur égal au nombre d'unités de calcul disponibles. En pratique, le gain est toujours moindre car il y a toujours des accès à des ressources communes n'acceptant pas plusieurs accès simultanés ou dont la bande passante ne permet pas d'alimenter toutes les unités de calcul en pleine vitesse.

12.3.2.2. Utilisation du multithreading

Classiquement, un programme monothreadé définit une unique liste des instructions qui doivent être exécutées par la machine. Cette liste est déterminée par un point d'entrée, généralement la fonction `main`. Dans les programmes multithreadés, le programme peut demander au système d'exploitation

la création d'autres flux d'exécution qui s'exécuteront de manière simultanée. Pour cela, il fournit au système l'adresse de fonctions qui serviront de point d'entrée aux différents threads, et dont l'exécution définira la liste d'instructions de ces threads. Les fonctions permettant de créer de nouveaux threads sont respectivement `_beginthreadex` sous Windows et `pthread_create` sous Unix et Linux. La documentation respective de ces environnements vous indiquera comment les utiliser.

Ainsi, un programme multithreadé peut exécuter plusieurs opérations de manière simultanée, et acquérir un comportement multitâche semblable à l'exécution simultanée de plusieurs processus dans les systèmes multitâches. La différence ici est que les différents threads appartiennent au même processus, et partagent donc le même espace d'adressage. Cela signifie que les données du programmes sont accédées par tous les threads du processus en parallèle. Les communications entre les threads sont donc plus faciles à mettre en œuvre et plus performantes que dans le cas de plusieurs processus qui s'exécutent de concert.

12.3.2.3. Les pièges du multithreading

Le multithreading est donc une technique intéressante quasiment incontournable dans tous les programmes non triviaux. Cependant, il apporte également son lot de problèmes, et provoque l'apparition de nouveaux types de bogues, dont les principaux sont les suivants :

- les concurrences d'accès ;
- les indéterminismes ;
- les interblocages.

Les concurrences d'accès sont dues au fait même que les données du programme sont accessibles de l'ensemble des threads. Ainsi, si un thread modifie les données pendant qu'un ou plusieurs autres threads cherchent à les lire, les données lues risquent d'être incohérentes. Par exemple, le début des données lues peut correspondre aux données avant modification par le thread écrivain, et la fin des données aux données après modification. De même, si plusieurs threads cherchent à écrire des valeurs différentes en mémoire, le résultat est généralement indéterminé.

Ces concurrences d'accès sont généralement éliminées en utilisant des primitives d'exclusion mutuelle qui s'apparentent à des verrous. Chaque thread devant accéder aux données partagées doit au préalable acquérir le verrou, qui est une ressource gérée par le système qui garantit qu'un seul thread à un instant donné peut en disposer (voir la fonction `EnterCriticalSection` sous Windows et la fonction `pthread_mutex_lock` sur les systèmes Posix). Si un autre thread dispose du verrou, le thread appelant est bloqué (c'est-à-dire que son exécution est suspendue) jusqu'à ce que le détenteur du verrou le relâche. Les portions de code exécutées avec un verrou pris sont classiquement appelées des « sections critiques ».

Les indéterminismes sont une généralisation des concurrences d'accès et consistent en la production de résultats différents à chaque exécution du programme, en raison des différences de vitesse d'exécution des différents threads qui le constituent. Par exemple, un thread peut utiliser un algorithme qui effectue deux traitements différents en fonction de la valeur d'une variable, et un autre thread peut modifier cette valeur de manière simultanée. Même si les accès à la variable sont contrôlées par un verrou, le comportement du premier thread peut être différent selon que le deuxième thread a eu le temps ou non de modifier la variable.

Les indéterminismes sont généralement éliminés en utilisant des primitives de synchronisation entre threads (voir les fonctions `CreateEvent`, `WaitForSingleObject` et `SetEvent` sous Windows, et les fonctions `pthread_cond_init`, `pthread_cond_wait` et `pthread_cond_signal` sur les systèmes Posix). Dans notre exemple, le premier thread peut attendre que le deuxième ait modifié

la variable. C'est donc au deuxième thread de signaler qu'il l'a fait, en indiquant au système que la condition qu'attend le premier thread est vérifiée.

Les interblocages sont caractérisés par le fait qu'un ou plusieurs thread restent suspendus de manière permanente, en attendant une ressource qui ne peut se libérer ou une condition qui ne peut se produire, car le thread qui est capable de lever la condition de la suspension est lui-même en attente d'une autre ressource ou d'une autre condition dépendant du premier thread. Le cas d'école classique est celui où deux threads veulent accéder à deux verrous différents, et que chacun d'eux ont pris l'accès à l'un des verrous et attend que l'autre thread relâche l'autre.

Il y a deux techniques fondamentales qui permettent d'éliminer les interblocages :

- réduire la portée des sections critiques au code qui accède réellement aux données partagées par les threads ;
- s'assurer que les verrous pris successivement le sont toujours dans le même ordre par tous les threads du programme.

Ces deux règles ont un impact fort sur la conception des programmes orientés objets.

> **Note :** On notera qu'un thread peut s'autobloquer (par exemple en attendant la fin d'un traitement et que ce traitement est à sa charge). Il n'est donc pas nécessaire, techniquement parlant, de disposer de plusieurs threads pour provoquer un blocage.

12.3.2.4. Multithreading et programmation objet

Le multithreading est relativement difficile à mettre en œuvre dans le contexte de la programmation objet, car les données membres des objets doivent être protégées contre les accès multithreadés. Pratiquement, il faut s'assurer que ces données membres sont bien protégées quelles que soient les méthodes et les interfaces utilisées, d'une part, et qu'elles sont toutes cohérentes à tout instant, d'autre part.

La technique la plus classique est de définir un verrou pour l'objet et de le prendre pendant l'exécution de chaque méthode de l'objet. Cette technique convient parfaitement dans le cadre d'objets simples, mais peut provoquer des interblocages dans le cas d'interactions complexes avec d'autres objets. En effet, si l'objet utilise un autre objet disposant lui-même d'un verrou, il y a un fort risque que plusieurs threads réalisent un interblocage en utilisant ces objets de manière simultanée (en accédant aux verrous de ces objets dans un ordre différent). Comme il est assez difficile de déterminer, à la simple lecture de l'interface d'un objet, quels sont les autres objets qu'il va utiliser, la programmation multithreadée en environnement objet est extrêmement technique.

De plus, dans les systèmes à composants, ou dans les programmes utilisant des plugins, certains objets utilisés peuvent avoir été développés sans tenir compte du multithreading. De ce fait, ces objets ne peuvent être accédés que par un unique thread durant toute leur durée de vie. Cela implique la mise en place de threads de travail pour certains objets et de mécanismes de changement de contexte pour l'exécution des appels de méthodes par ces threads de travail. Classiquement, cela conduit à la définition de « modèles de threading » pour les objets et d'« appartements » dans lesquels les objets d'un certain modèle sont cloisonnés afin de garantir une utilisation correcte (ces notions sont particulièrement visibles dans les systèmes à composants tels que la technologie COM de Microsoft ou son concurrent CORBA).

Malheureusement, ces notions sont complexes et imposent des restrictions fortes, très souvent mal maîtrisées par la plupart des programmeurs. De plus, elles ne peuvent être prises en compte a posteriori, car elles imposent systématiquement une architecture particulière pour les programmes, et les

bogues liés aux modèles de threading ne peuvent généralement pas se corriger sans une modification structurelle importante du programme.

Par conséquent, et là je pèse mes mots, il est extrêmement important de prendre en compte les problèmes relatifs au multithreading dès la conception du logiciel. Plus que jamais, en environnement multithreadé, les recommandations suivantes devront être respectées :

- il faut parfaitement identifier les objets manipulés par le programme ;
- il faut parfaitement identifier le contexte d'utilisation de ces objets ;
- il faut parfaitement définir les interfaces utilisables ;
- il faut parfaitement définir la manière d'utiliser ces interfaces et la dynamique du système.

La première règle est un prérequis à la deuxième et est imposé par la méthode de conception objet. La deuxième règle a pour but de forcer précisément la définition des threads susceptibles d'utiliser les objets, afin d'identifier les données à protéger et les verrous à mettre en place, ainsi que les modèles de threading des objets. La troisième règle permet de contrôler les points d'entrée des threads. Enfin, la dernière permet de déterminer la liste des ressources utilisées, comment les objets seront utilisés, et où les changements de contexte se feront.

12.3.2.5. Limitations et contraintes liées au multithreading

Si le multithreading peut s'avérer utile, il ne faut pas en abuser non plus. En effet, les threads restent des ressources systèmes relativement lourdes, et ont malgré tout un certain coût. Premièrement, ils induisent une surcharge du système à cause des changements de contexte. En effet, même si les machines modernes sont capables de changer de thread rapidement, les caches d'instructions du processeur sont malgré tout invalidés lorsqu'un tel changement se produit. Il est à noter que le multithreading est sur ce point malgré tout préférable au multiprocessus, puisqu'un changement de contexte interprocessus implique un changement d'espace d'adressage et donc également l'invalidation de tous les caches système.

De plus, chaque thread dispose d'une pile d'exécution, afin de mémoriser les variables locales et les appels des fonctions qu'il exécute. Cette pile d'appel consomme évidemment de la mémoire, et même si elle est agrandie à la volée par le système, elle implique de réserver une plage d'adresses dans le processus. La création d'un grand nombre de threads implique donc une fragmentation de l'espace d'adressage, qui peut ensuite faire échouer les allocations dynamique de mémoire.

À titre d'exemple, la taille réservée dans l'espace d'adressage pour la pile d'un thread est généralement de un mégaoctet (attention, il s'agit bien d'une plage d'adresses réservées et non de mémoire effectivement allouée). Sur les systèmes trente-deux bits, l'espace d'adressage fait quatre gigaoctets, ce qui limite de facto le nombre de threads à quatre mille. De plus, l'espace d'adressage des processus étant généralement divisé en deux parties (la partie supérieure étant utilisée par le système pour y placer une image de la mémoire du noyau), le nombre maximal est en pratique souvent inférieur à deux mille. Pour aller au delà, il faut réduire la taille des piles des threads, mais il serait plus sain dans ce type de situation de se demander si le programme est bien conçu...

Une autre restriction courante dans les programmes multithreadés est que les bibliothèques et les environnement graphiques exigent, souvent pour des raisons de compatibilité avec d'anciennes bibliothèques système ou avec des composants graphiques monothreadés, que toutes les opérations relatives à l'interface homme-machine soient effectuées sur le thread principal (c'est-à-dire le thread qui exécute la fonction `main`). En pratique donc, il est recommandé que ce thread ne prenne en charge que les opérations de gestion de l'interface graphique, et délègue tous les traitements à des threads de

travail. C'est une contrainte à prendre en compte au niveau de la conception même du logiciel. Il faut donc établir des mécanismes de communication entre les threads, qui permettent au thread principal de poster une demande d'exécution d'une commande à un thread de travail et d'en attendre le résultat, tout en continuant à gérer l'interface graphique (affichage d'un sablier, redessin de la fenêtre, etc.).

12.3.3. Les signaux

Les signaux sont des mécanismes utilisés pour interrompre le flux d'exécution des programmes et leur indiquer un événement particulier. Le principe des signaux prend racine dans les fondements d'Unix et fournit une forme de communication inter-processus rudimentaire.

Lorsqu'un programme reçoit un signal (soit de la part du système, soit de la part d'un autre programme, comme la commande kill par exemple), son exécution est interrompue et une fonction particulière appelée gestionnaire du signal est exécutée. Généralement, si le programme n'a défini aucun gestionnaire de signal ou n'a pas demandé à les masquer, le système l'arrête brutalement. Si en revanche ce gestionnaire existe, il est exécuté et le processus normal du programme reprend son cours.

Les signaux fournissent donc un mécanisme semblable aux interruptions pour les systèmes d'exploitation. Cependant, ils souffrent de défauts conséquents, qui rendent leur emploi difficile :

- il existe deux API, l'une spécifiée dans la norme C mais très peu pratique, l'autre spécifiée dans la norme POSIX, mais non portable ;
- l'API ANSI C n'a pas été interprétée de la même manière par tous les systèmes et n'est donc pas portable (selon l'implémentation, les gestionnaires de signaux doivent être réinstallés ou non après chaque occurrence du signal) ;
- les signaux peuvent se produire de manière asynchrone, donc virtuellement n'importe quand, et le programme doit être en mesure de pouvoir les traiter en permanence ;
- de ce fait, les gestionnaires de signaux ne peuvent accéder à quasiment aucune ressource du programme (en particulier, il est interdit d'appeler les fonctions de la bibliothèque C, pas même les fonctions `malloc`/`free` ou `printf`) ;
- selon la manière dont ils sont utilisés, les signaux peuvent interrompre les appels systèmes, forçant ainsi le programmeur à effectuer tous les appels systèmes dans une boucle pour les réessayer en cas d'apparition d'un signal ;
- les signaux ne se comportent pas de manière déterministe dans les programmes multithreadés, car ils peuvent être délivrés à un thread quelconque du programme.

Les signaux sont donc extrêmement techniques à manipuler. En pratique, ils ne sont utiles que pour les programmes extrêmement simples. Dans les autres programmes, il est plus simple d'adopter une des méthodes suivantes :

- soit les signaux sont purement et simplement masqués ;
- soit le programme s'interrompt brutalement dès la réception d'un signal ;
- soit les signaux sont traités dans un thread dédié à cette tâche.

Cette dernière solution se met classiquement en place en bloquant tous les signaux dans le thread principal avec la fonction `pthread_sigmask` avant de créer les autres threads du programme, puis en créant le thread dédié à la gestion des signaux. Le rôle de ce thread est ensuite de récupérer tous

les signaux en boucle à l'aide de la fonction `sigwait`. Consultez la documentation des spécifications Unix unifiées pour plus de détails sur ces fonctions.

> **Note :** Les signaux sont disponibles sous Windows, mais sont simulés par des appels de fonction de rappel sur des threads créés pour l'occasion. Leur emploi est fortement déconseillé, car les mécanismes des signaux ne sont absolument pas utiles et implémentés a minima sous Windows.

12.3.4. Les bibliothèques de liaison dynamique

Les bibliothèques de liaison dynamique (« DLL », en anglais, abréviation de « Dynamic Link Library ») sont des bibliothèques de programme qui ne sont pas intégrées directement dans le fichier exécutable du programme. Cela signifie que l'accès à leurs fonctionnalités ne peut se faire de manière directe, et nécessite une opération préalable de chargement et de récupération des symboles de la bibliothèque.

Les bibliothèques de liaison dynamique sont donc des bibliothèques qui sont chargées à la demande par les programmes qui désirent les utiliser. De ce fait, elles permettent de réaliser des choses que les bibliothèques de liaison statique n'autorisent pas aussi facilement. Bien entendu, cela impose quelques contraintes supplémentaires, qu'il est important de connaître.

12.3.4.1. Les avantages des bibliothèques de liaison dynamique

Les principaux avantages de ces bibliothèques sont les suivants :

- elles peuvent être partagées entre plusieurs programmes, économisant ainsi espace disque et, parfois, espace mémoire (en permettant au système d'exploitation de factoriser le code de ces bibliothèques entre tous les programmes) ;
- elles permettent de ne charger les fonctionnalités demandées qu'à la demande, réduisant ainsi la consommation mémoire ;
- elles permettent de rendre les programme plus modulaire, en le découpant en un programme principal et en plusieurs bibliothèques de liaison dynamique secondaires.

Ce dernier point est sans doute le plus important, car il ouvre les possibilités suivantes :

- il est possible de réaliser des « bibliothèques de ressources », dont le but est de stocker des informations dépendantes du contexte (par exemple la langue des messages du programme) et de permettre d'en charger la version adaptée au contexte courant ;
- la mise à jour des programmes est simplifiée, car seuls les composants concernés par une correction de bogue doivent être mis à jour ;
- il est possible de rendre les programmes extensibles, en permettant le chargement dynamique de greffons (« plugins » en anglais) utilisés via des interfaces standardisées et en fonction de la configuration du programme.

12.3.4.2. Les mécanismes de chargement

Il existe classiquement deux manières d'accéder aux fonctions d'une bibliothèque de liaison dynamique.

La première méthode est totalement manuelle et fait appel à des fonctions du système qui permettent généralement les opérations suivantes :

- le chargement de la bibliothèque dans l'espace d'adressage du processus (fonctions `LoadLibrary` sous Windows et `dlopen` sous Unix ou Linux) ;
- la recherche d'un symbole (variable ou fonction) dans la bibliothèque à partir de son nom et l'obtention de son adresse (fonctions `GetProcAddress` sous Windows et `dlsym` sous Unix ou Linux) ;
- l'utilisation du symbole via le pointeur obtenu ;
- le déchargement de la bibliothèque une fois qu'elle n'est plus nécessaire (fonctions `FreeLibrary` sous Windows et `dlcose` sous Unix ou Linux).

Cette méthode est extrêmement lourde, car elle nécessite de manipuler les symboles par leur nom. Cela peut être relativement technique dans le cas des bibliothèques de liaison dynamique écrites en C++, en raison du fait que les noms des symboles sont décorés par les informations de typage du langage. Mais cette méthode est à la base du chargement des fonctionnalités à la volée par l'applicatif et permet la gestion des greffons de manière totalement dynamique (un greffon peut être installé et chargé pendant que le programme fonctionne).

Une autre méthode, généralement plus utilisée, est également disponible. Elle consiste simplement à réaliser l'édition de liens du programme avec la bibliothèque de liaison dynamique comme s'il s'agit d'une bibliothèque statique normale. Les symboles sont alors marqués comme étant non résolus dans le programme, et une table est ajoutée pour indiquer comment le système devra résoudre ces symboles à l'exécution. Ainsi, lorsque le programme s'exécutera, les symboles seront résolus dynamiquement, lors d'une phase appelée « édition de liens dynamique » (et dont le nom de ces bibliothèques provient).

Cette technique étant prise en charge automatiquement par le système d'exploitation et par le chargeur de programme, il est évident qu'elle est plus simple d'emploi. En particulier, il n'est plus nécessaire de connaître les noms des symboles, ce qui permet de faire des bibliothèques de liaison dynamique en C++ facilement. En revanche, elle impose de connaître l'ensemble des bibliothèques utilisées par le programme lors de son édition de liens, et ne permet donc pas de faire de bibliothèques de fonctions chargées à la demande par l'applicatif ni de faire des greffons.

L'opération de résolution des liens est relativement technique et dépend bien entendu beaucoup du système d'exploitation utilisé. En général, les liens qui doivent être résolus sont enregistrés dans une table stockée dans les fichiers binaires (la table des symboles importés), que le chargeur du système consulte et met à jour lors du chargement de ces fichiers. Tous les symboles importés par les programmes sont utilisés indirectement, via ces tables. Ainsi, une fois la résolution des liens effectuée, tous les symboles peuvent être accédés via une simple indirection.

12.3.4.3. Relogement et code indépendant de la position

L'un des plus gros problèmes des bibliothèques de liaison dynamique est qu'il n'est généralement pas possible, lors de leur création, de déterminer l'adresse à laquelle elles pourront être chargées en mémoire. Il est possible de fixer une adresse de chargement privilégiée, mais l'utilisation de cette adresse ne peut être garantie. En effet, lors du chargement, il se peut que le programme ait placé des

ressources à cette adresse ou qu'il n'y ait pas assez d'espace disponible pour charger la bibliothèque à cet endroit.

Selon le système d'exploitation et le processeur cible utilisé, la non-constance de l'adresse de chargement peut avoir des conséquences importantes sur le code généré par le compilateur ou les performances du programme. En effet, cela implique que tous les symboles de la bibliothèque ne peuvent être accédés avec une adresse absolue. Il faut nécessairement les référencer par une adresse définie par rapport à l'adresse de chargement de la bibliothèque. Comme cette adresse ne peut être connue avant le chargement de la bibliothèque, et comme on ne peut imposer de manière sûre l'adresse de chargement de la bibliothèque, le code généré par le compilateur doit être modifié pour chaque utilisation d'un symbole de la bibliothèque, que ce soit une variable ou une fonction !

Cette opération, appelée « relogement » (« relocation » en anglais), est réalisée soit par le compilateur, soit par le chargeur du système.

Sous Windows, c'est le chargeur qui s'en occupe. Toutes les bibliothèques de liaison dynamique Windows disposent d'une table des relogements à effectuer une fois la bibliothèque chargée en mémoire. Cette table indique toutes les instructions du code de la bibliothèque qui doivent être corrigées en fonction de l'adresse effectivement utilisée pour charger la bibliothèque. La correction consiste simplement en une simple addition de cette adresse à chaque offset utilisé par les instructions déclarées dans la table des relogements. Ainsi, une fois le relogement effectué, le code de la bibliothèque est corrigé pour accéder à ses données, comme si le compilateur avait deviné l'adresse de chargement de la bibliothèque lors de sa création.

Inversement, sous les systèmes Unix et Linux, le code généré par le compilateur peut être spécialement écrit de manière à ne jamais faire de référence directe aux symboles. Un tel code est dit « indépendant de sa position » (« Position Independent Code » en anglais). Pratiquement, les adresses des symboles sont récupérées dans une table globale de la bibliothèque (appelée la « Global Offset Table »). Cette table est mise à jour lors du chargement de la bibliothèque. Mais cela ne suffit pas : pour y accéder, le code de la bibliothèque doit en plus conserver l'adresse de cette table dans toutes les fonctions, ce qui se fait généralement en réservant un registre du processeur à cet effet ou en la stockant dans une mémoire accessible directement (par exemple sur la pile).

> **Note :** Si la technique utilisée par les systèmes Unix est relativement lourde et peut avoir un léger impact sur les performances du code à l'exécution, celle utilisée par Windows impose la modification du code des bibliothèques. De ce fait, ce code ne peut généralement plus être partagé entre plusieurs processus, et une consommation mémoire accrue du système s'ensuit. Cela n'est toutefois pas dérangeant pour les bibliothèques spécifiques à un programme, qui n'ont pas été crées pour partager le code mais dans le but de bénéficier de la souplesse de déploiement ou des mécanismes de plugins.
>
> Les chargeurs de programmes Unix sont capables d'effectuer le relogement des symboles dans les bibliothèques dont le code n'est pas compilé de manière à être indépendant de la position. Cependant, cette technique cela n'est pas recommandé, en raison du fait que le code de ces bibliothèques ne peut plus être partagé entre les processus.

12.3.4.4. Optimisation des bibliothèques de liaison dynamique

Comme on l'a vu, les bibliothèques de liaison dynamique sont très pratiques, mais imposent un travail supplémentaire au chargeur du système d'exploitation lors de leur chargement. De plus, sur les plateformes Unix et Linux, l'accès aux symboles exportés par les bibliothèques n'est généralement pas direct.

Tout cela se traduit par un temps de chargement des programmes plus long, et parfois une exécution légèrement plus lente. Si cela n'est pas gênant pour les petits programmes, ce surcoût peut devenir conséquent pour les grands programmes, qui utilisent de nombreuses bibliothèques pouvant exporter des milliers de symboles. La situation est critique pour les programmes C++ sous Unix et Linux car, par défaut, tous les symboles sont exportés (jusqu'au moindre accesseur de la moindre classe).

Toutefois, si le compilateur peut savoir, dès la génération du code, qu'un symbole ne sera pas exporté par la bibliothèque, il peut réaliser des optimisations. En effet, il peut dans ce cas faire en sorte que les instructions qui y accèdent le fasse de manière relative à leur adresse. Notez cependant que tous les processeurs ne disposent pas forcément des modes d'adressage nécessaires pour cela (par exemple, les processeurs x86 ne permettent pas d'accéder aux données relativement à l'adresse du pointeur d'instruction, par contre, ils sont tout à fait capables de faire des appels de fonctions et des sauts relatifs à l'adresse de l'instruction courante). Dans ce cas, les symboles peuvent malgré tout être référencés relativement à l'adresse de chargement de la bibliothèque, ce qui évite de passer par la table des offsets globaux.

La meilleure manière d'optimiser les programmes qui utilisent intensivement les bibliothèques de liaison dynamique est donc de réduire le nombre de symboles exportés au strict nécessaire, c'est-à-dire à l'interface publique des bibliothèques. Cette technique permet de plus de mieux encapsuler les structures de données des bibliothèques, et est donc intéressante à double titre.

Il est donc intéressant de pouvoir spécifier explicitement quels sont les symboles exportés directement au niveau du code source. Cela permet en effet :

- de réduire la taille de la table des symboles exportés, et d'accélérer leur recherche lors de la résolution des liens dynamiques ;
- de réduire la taille de la table des offsets des symboles globaux de la bibliothèque, et d'accélérer le chargement de la bibliothèque et d'optimiser le code en réduisant le nombres de symboles accédés via cette table.

> **Note :** La réduction du nombre de symboles exportés n'a toutefois aucun impact sur la taille des tables de relogement des bibliothèques Windows ou des bibliothèques Unix/Linux dont le code n'est pas indépendant de sa position. Le temps de l'opération de relogement et le taux des pages mémoire contenant un code non modifié, et donc partageable entre processus, sont donc inchangés.

Par défaut, les bibliothèques de liaison dynamiques Windows n'exportent aucun symbole. La liste des symboles exportés doit donc être fournie à l'éditeur de liens lors de la création de la bibliothèque. Cela se fait généralement en lui fournissant d'un fichier de définition des symboles exportés sur sa ligne de commande (ces fichiers portent généralement l'extension « `.def` »).

Sous Unix et Linux, tous les symboles externes sont exportés par défaut. Il possible de modifier la nature des symboles lors de l'édition de liens, en spécifiant, encore une fois, la liste des symboles exportés à l'éditeur de liens. Toutefois, la manière de procéder pour définir les symboles exportés et en donner la liste à l'éditeur de liens est extrêmement technique. De plus, cette technique ne permet pas au compilateur d'optimiser le code.

Cette solution n'est par ailleurs pas pratique du tout dans le cas des bibliothèques de liaison dynamique écrites en C++, du fait que la liste des symboles peut ne pas être connue puisque leurs noms sont décorés par le compilateur. Par conséquent, la seule vraie solution est de définir les symboles exportés directement au niveau du code source, par une construction syntaxique dédiée à cet effet.

Or,le C standard ne dispose que de peu de constructions syntaxiques pour cela. En pratique, seul le mot clef `static` permet, par définition, de ne pas exporter un symbole. Malheureusement, il n'y a pas de solution standard permettant de ne pas exporter un symbole non statique, et il est rare que l'on n'utilise qu'un seul fichier pour coder une bibliothèque !

Des extensions ont donc été ajoutées aux compilateurs afin de spécifier quels sont les symboles qui doivent être exportés et quels symboles sont uniquement utilisés par la bibliothèque. Les mécanismes utilisés sont relativement semblables sous Windows et sous Unix et Linux, et consistent simplement à appliquer un attribut aux symboles exportés lors de leur déclaration.

Ainsi, sous Windows, si aucun fichier d'export n'est fourni à l'éditeur de liens, seuls les symboles déclarés avec l'attribut « `dllexport` » sont exportés :

```
extern int __declspec(dllexport) global;
extern int __declspec(dllexport) get_new_int();
extern int private_fuction(int);
```

Sous Unix et Linux, les symboles disposent d'un attribut de visibilité, permettant de spécifier la portée de ces symboles lors de l'édition de liens. La valeur « `default` » de cet attribut correspond à l'usage classique des symboles : ils sont exportés s'ils sont externes, et sont privés s'ils sont statiques. L'attribut de visibilité peut aussi prendre la valeur « `hidden` », ce qui a pour effet de masquer ces symboles une fois l'édition de liens réalisée. Le compilateur peut également réaliser les optimisations lors de la génération du code indépendant de sa position. Avec le compilateur GCC, la valeur de l'attribut de visibilité peut être fixée avec la syntaxe suivante :

```
extern int __attribute__((visibility("default"))) global;
extern int __attribute__((visibility("default"))) get_new_int();
extern int __attribute__((visibility("hidden"))) private_fuction(int);
```

Les symboles dont la visibilité n'est pas explicitement spécifiée prendront la visibilité spécifiée par l'option « `-fvisibility` » de GCC. Si cette option n'est elle-même pas spécifiée, la valeur par défaut de l'attribut de visibilité est « `default` ». Par conséquent, il est recommandé de marquer explicitement les symboles à exporter avec la visibilité « `default` » à l'aide d'un attribut, et de forcer la visibilité par défaut des autres symboles à « `hidden` » à l'aide de l'option « `-fvisibility` ».

> **Note :** Sous Windows, les symboles importés doivent être déclarés avec l'attribut « `dllimport` ». Par conséquent, les fichiers d'en-tête doivent fournir deux déclarations pour les symboles des bibliothèques de liaison dynamique, une avec l'attribut `dllexport`, qui sera utilisée par la bibliothèque elle-même, et une avec l'attribut `dllimport`, qui sera utilisée par les programmes clients. Il est courant de définir une macro `DLLSYMBOL` conditionnée par une option de compilation fournie en ligne de commande et dépendant du contexte d'utilisation du fichier d'en-tête :
>
> ```
> /* Définition de la macro DLLSYMBOL : */
> #if defined(MA_DLL)
>    /* Utilisation dans la DLL MA_DLL, compilée avec l'option
>       MA_DLL (définie sur la ligne de commande du compilateur) : */
>    #define DLLSYMBOL __declspec(dllexport)
> #else
>    /* Utilisation dans un programme client : */
>    #define  DLLSYMBOL __declspec(dllimport)
> #endif
>
> /* Définition des symboles de la DLL : */
> ```

```
int DLLSYMBOL global;
int DLLSYMBOL get_new_int();
```

Cette technique peut également être utilisée pour simplifier la spécification de la visibilité des objets sous Unix et Linux.

12.3.4.5. Initialisation des bibliothèques de liaison dynamique

Les bibliothèques de liaison dynamique peuvent définir des fonctions d'initialisation et de nettoyage, que le chargeur du système peut appeler respectivement après le chargement et avant le déchargement des bibliothèques de liaison dynamique. La manière dont ces fonctions sont définies dépend du système d'exploitation. En général, ces informations sont fournies à l'éditeur de liens statiques lors de la création du programme.

Classiquement, le point d'entrée des bibliothèques de liaison dynamique sous Windows se nomme `DllMain` :

```
BOOL DllMain(HINSTANCE, DWORD dwReaon, LPVOID lpvReserved);
```

Cette fonction est appelée lors du chargement et lors du déchargement de la bibliothèque dans le processus et, sauf configuration explicite, lors de la création et lors de la terminaison d'un thread dans le processus. La raison de l'appel est indiquée par un paramètre de la fonction. L'appel se fait dans le contexte du thread concerné par l'appel. Autrement dit, les appels indiquant que la bibliothèque est chargée ou déchargée sont réalisés dans le contexte du thread qui effectue le chargement ou le déchargement, et les appels indiquant qu'un thread est créé ou se termine sont faits dans le contexte de ce thread. On notera que les initialisations relatives aux threads ne sont réalisées que pour les threads qui se créent ou se détruisent après que la bibliothèque a été chargée. `DllMain` n'est donc pas appelée pour les threads déjà existants dans le processus lors du chargement ou du déchargement. Il est donc vivement déconseillé de réaliser un traitement spécifique aux threads dans cette méthode, car il ne peut être réalisée de manière symétrique au chargement et au déchargement.

Sous Unix et Linux, les mécanismes ne sont pas spécifiés de manière standard. Si l'on utilise le compilateur GCC, il est possible de définir deux fonctions pour l'initialisation et la libération des ressources, qui seront appelées dans le contexte du chargement et du déchargement (contrairement à Windows, il n'y a pas de mécanismes pour prévenir une bibliothèque de l'apparition ou de la disparition d'un thread une fois celle-ci chargée). Ces deux fonctions doivent être identifiées respectivement à l'aide des attributs spéciaux « `__attribute__((constructor))` » et « `__attribute__((destructor))` » :

```
void __attribute__((constructor)) init_lib(void);
void __attribute__((destructor)) release_lib(void);
```

Quel que soit le système utilisé et pour quelque raison que ce soit, l'utilisation des fonction d'initialisation et de libération des bibliothèques devra se faire avec la plus grande précaution. En effet, le chargement d'une bibliothèque dans un processus est un événement très particulier dans la vie du processus. Du fait que ces opérations modifient l'espace d'adressage du processus, elles se font dans un contexte extrêmement sensible. Généralement, l'appel de ces fonctions se fait souvent

au sein d'une section critique. Tous les threads de l'application peuvent également être suspendus, sauf bien entendu le thread qui effectue le chargement ou le déchargement. Les ressources des autres bibliothèques peuvent ne pas être accessibles, ainsi même que la plupart des ressources de la bibliothèque elle-même. Par conséquent, ces fonctions ne peuvent réaliser que des opérations extrêmement simples. Toute opération complexe (allocation dynamique de mémoire, lancement d'exception, création de thread, synchronisation, etc.) sont absolument interdites.

Les objets globaux constituent un cas particulier important. Ces objets sont toujours initialisés, normalement dans l'ordre de leurs définitions, par le thread qui effectue le chargement du module qui les contient, et avant l'exécution du point d'entrée. Par exemple, les constructeurs des objets statiques des programmes sont appelés dans le contexte du thread principal de l'application, avant l'exécution de la fonction `main`. Dans le cas des bibliothèques de liaison dynamique, ils sont appelés avant l'appel des fonctions d'initialisation de ces bibliothèques. Il va de soi que ces constructeurs ne peuvent pas réaliser d'opérations sensibles, et doivent se conformer aux mêmes règles que celles imposées aux fonctions d'initialisation et de libération des ressources des bibliothèques.

12.3.5. Les communications réseau

Les communications réseau requièrent une attention toute particulière. Cela est du au fait que les informations peuvent être non fiables, que les temps d'accès aux informations sont plus lents, et que les liaisons peuvent être coupées. De plus, les protocoles réseau peuvent révéler des pièges qu'il est préférable de connaître dès la conception du logiciel.

12.3.5.1. Fiabilité des informations

En général, les données qui proviennent d'une communication ne peuvent pas être considérées comme fiables. En effet, les réseaux ne sont pas conçus pour transmettre des informations sans dégradation des données.

Il est donc nécessaire de mettre en place des mécanismes d'accusé réception, de contrôle d'intégrité des données, et de réémission en cas de détection d'une erreur ou de la perte d'une information. Avec ces mécanismes, les interlocuteurs peuvent s'assurer qu'ils obtiennent bien les données et que ce sont les bonnes.

Cependant, ces mécanismes ne sont implémentés que par certains protocoles (notamment le protocole de transmission TCP/IP utilisé sur Internet), mais pas par tous (par exemple, le protocole d'échange de datagrammes UDP/IP et IP lui-même ne garantissent ni la délivraison, ni l'intégrité des informations transmises).

Mais même lorsqu'un protocole supposé fiable comme TCP/IP est utilisé, les données ne peuvent pas être considérées comme sûres. En effet, les mécanismes de vérification d'intégrité de la plupart des protocoles se basent sur des sommes de contrôle sur les données transmises. Or des paquets différents peuvent parfaitement produire une somme de contrôle identique, bien que cela soit très peu probable si la différence est due à une simple erreur de communication. En particulier, les mécanismes de type CRC, souvent utilisés, n'ont pas été conçus pour garantir qu'une donnée est intègre, mais pour permettre de la restaurer lorsqu'une erreur simple s'est produite ! Pour garantir l'intégrité des données transmises, il faudrait utiliser des algorithmes de calcul de condensés cryptographiques.

De plus, un paquet parfaitement valide, transmis sans perte de données, peut ne pas être un paquet correct pour l'application. En effet, il est parfaitement possible de forger des paquets tout à fait valides, mais construits spécialement pour faire planter une application ou en prendre le contrôle. Il s'agit bien entendu là de piratage, mais toute application communiquant via un réseau non protégé avec un protocole non chiffré se doit de prendre en considération ce cas de figure.

Il est donc particulièrement important, lors de la spécification des protocoles réseau applicatifs, ainsi que lors de la définition des messages échangés, de s'assurer que l'application peut contrôler la validité des informations. Cela signifie en particulier qu'elle ne doit en aucun cas prendre pour argent comptant les données qui proviennent de l'extérieur (règle de validation des entrées), et que les messages échangés doivent contenir toutes les informations nécessaires à leur interprétation sans avoir recours à des hypothèses externes.

Ainsi, tout compteur, tout indice et toute référence doivent pouvoir être vérifiés, et l'être. Par exemple, la détermination de la longueur des données ne doit pas se faire sur la seule information de la quantité d'informations reçues du réseau. Inversement, si un champ longueur est transmis dans un message, il faut s'assurer que la taille des données du message est suffisante pour en interpréter la suite.

12.3.5.2. Performances des communications

Un autre aspect important des communications réseau est qu'elles sont beaucoup plus lentes que les autres traitements que les programmes peuvent réaliser. De ce fait, ce facteur doit être pris en compte lors de la conception des logiciels et des protocoles réseau.

Dans un premier temps, les logiciels ont intérêt à ne pas se bloquer en attendant que les opérations réseau se terminent. Le minimum est de fournir à l'utilisateur un signe de vie et, lorsqu'une opération longue est en cours, de le lui indiquer. Il est rarement possible de déterminer la durée d'une communication réseau, car les conditions de débit peuvent changer dans le temps suivant la charge des réseaux traversés par le flux d'information, mais il est souvent possible d'indiquer la quantité de travail restant à faire (par exemple, la taille des données restant à recevoir). Enfin, idéalement, donner à l'utilisateur la possibilité de faire d'autres opérations pendant que les communications se font est un plus incontestable.

Du point de vue des communications elles-mêmes à présent, il faut s'assurer que les échanges soient réduits au strict minimum. En effet, chaque requête et chaque réponse sont soumises aux règles de gestion du protocole de communication sous-jacent, et donc aux mécanismes d'accusés réception et de ré-émissions en cas d'erreur. De plus, certains mécanismes d'optimisation des protocoles peuvent amplifier les latences lors des échanges, au profit de la bande passante. Ces mécanismes ont donc pour effet de bord que le temps de traitement des requêtes réseau est considérablement plus important que le temps de transfert des informations elles-mêmes. De ce fait, il est généralement plus rapide d'envoyer une requête complexe que de multiples requêtes simples.

Par conséquent, il est recommandé de privilégier des méthodes plus complexes au niveau des interfaces logicielles, mais qui sont adaptées aux besoins des clients et qui leur évite ainsi de faire d'autres requêtes. De même, il est recommandé de grouper les informations lors des communications, quitte à en envoyer quelquefois plus que nécessaire.

À titre d'exemple, le protocole TCP/IP cherche, par défaut, à augmenter la bande passante en réduisant la proportion des informations du protocole (en-têtes des paquets TCP et IP) par rapport aux informations utiles (données utilisées par l'application). Pour cela, il utilise un tampon pour les données en sortie, et n'envoie ces données que lorsqu'une quantité suffisante est disponible. Cela implique qu'il n'envoie pas les données immédiatement, mais temporise leur émission afin de pouvoir les envoyer de manière groupée. Bien entendu, s'il n'y a pas d'autres données, il les enverra malgré tout, mais après expiration d'un délai. De ce fait, si une application cherche à émettre des petits paquets en grande quantité, TCP/IP optimisera les communications pour elle.

Cela ne fonctionne toutefois pas dans un cas : lorsque l'application attend, pour envoyer les données suivantes, une réponse de la part de son interlocuteur. Dans ce cas, les deux programmes attendront systématiquement deux fois le délai d'émission du protocole TCP/IP. Il est possible de fixer les options sur les canaux de communication qui permettent de désactiver cette fonctionnalité de TCP/IP (option `TCP_NODELAY`), mais cela peut ne pas fonctionner. En effet, les protocoles réseau peuvent

être encapsulés (par exemple dans un tunnel HTTP pour passer au travers d'un pare-feu), et cette encapsulation n'est a priori pas détectable et encore moins configurable. Il est donc préférable dans ce cas de revoir le protocole de communication pour émettre les informations de manière groupée et recevoir ensuite les réponses en bloc, quitte à ce que certaines de ces réponses soient « je n'avais pas besoin de cette donnée ».

12.3.5.3. Pertes de connexion

Qui dit réseau dit perte de connexion. Les programmes qui communiquent en réseau doivent prendre en compte l'impossibilité de se connecter et la possibilité d'une perte de connexion, et ce quasiment à tous les niveaux. Cela va jusqu'à l'utilisateur final, qui doit pouvoir être prévenu qu'une erreur éventuellement irrécupérable s'est produite (soit par un message, soit par une trace).

Prendre en compte les pertes de connexion peut être difficile, mais les détecter l'est encore plus. Les pertes de connexion se détectent généralement par le fait qu'après un certain nombre de tentatives, les informations n'ont toujours par été reçues par l'interlocuteur (ou, qu'inversement, après un certain délai, aucune information n'a été reçue). La détection des pertes de connexion implique donc une notion de délai pendant lequel aucune donnée ne peut être échangée. Cela signifie qu'il n'est pratiquement pas possible de garantir à la fois la fiabilité des communications et une grande réactivité dans la détection des pertes de connexion.

En particulier, les protocoles fiables comme TCP/IP peuvent utiliser des durées très longues avant de se résoudre à déclarer forfait et à signaler une perte de connexion. L'utilisation de ces protocoles impose donc d'accepter cet état de fait.

Par exemple, en raison des mécanismes d'optimisation et de ré-émission, il n'est pas possible de savoir si une information a été effectivement transmise ou non lorsqu'il y a une perte de communication sur un canal TCP/IP. Du fait de l'utilisation d'un tampon de sortie, les écritures se font toujours dans le tampon, et lorsque la liaison est coupée, on ne peut pas savoir si les données sont encore dans le tampon ou si elles ont atteint l'autre bout du canal. De plus, même en cas de coupure de connexion, il reste encore possible d'envoyer des données dans le tampon de sortie, et ce pendant une durée a priori très longue, jusqu'à ce que TCP/IP abandonne les ré-émissions des informations. Si la perte de connexion est avérée, il est tout à fait possible que le programme ne s'en rende compte que très tardivement... Il est même possible qu'il ne s'en rende compte jamais, s'il a fermé le canal !

Il est donc particulièrement important, lors de la conception d'un programme réseau, de choisir les protocoles réseau utilisés en fonction des besoins de fiabilité, de performances et de réactivité.

> **Note :** Il est classique de mettre en place des mécanismes de chiens de garde pour surveiller les liaisons pendant toute la durée des connexions. Ces chiens de garde fonctionnent simplement en envoyant un message auquel l'interlocuteur doit répondre dans un temps imparti. Si cela n'est pas vérifié, la liaison peut être déclarée coupée.
>
> Cette technique est tout à fait valable, et permet au moins un contrôle applicatif de l'état des liaisons réseau. Toutefois, elle peut également se révéler difficile à mettre en place. En particulier, si les threads qui prennent en charge les messages de chien de garde ont d'autre tâches à effectuer, ou si le canal utilisé pour le chien de garde peut être utilisé pour transférer de grandes quantités d'information, le mécanisme devient sensible à la charge des systèmes à surveiller. Dans ce cas, les chiens de garde ne contrôlent donc plus uniquement la liaison, mais aussi la disponibilité des applications. En cas de charge de travail importante, ce qui se produit notamment lors du démarrage et lors des connexions initiales, le chien de garde peut se déclencher de manière intempestive et induire un travail supplémentaire indésirable. Si le traitement du chien de garde est une reconnexion, le système peut ne pas arriver à se stabiliser.

II. La bibliothèque standard C++

Tout comme pour le langage C, pour lequel un certain nombre de fonctions ont été définies et standardisées et constituent la bibliothèque C, une bibliothèque de classes et de fonctions a été spécifiée pour le langage C++. Cette bibliothèque est le résultat de l'évolution de plusieurs bibliothèques, parfois développées indépendamment par plusieurs fournisseurs d'environnements C++, qui ont été fusionnées et normalisées afin de garantir la portabilité des programmes qui les utilisent. Une des principales briques de cette bibliothèque est sans aucun doute la STL (abréviation de « Standard Template Library »), à tel point qu'il y a souvent confusion entre les deux.

Cette partie a pour but de présenter les principales fonctionnalités de la bibliothèque standard C++. Bien entendu, il est hors de question de décrire complètement chaque fonction ou chaque détail du fonctionnement de la bibliothèque standard, car cela rendrait illisibles et incompréhensibles les explications. Cependant, les informations de base vous seront données afin de vous permettre d'utiliser efficacement la bibliothèque standard C++ et de comprendre les fonctionnalités les plus avancées lorsque vous vous y intéresserez.

La bibliothèque standard C++ est réellement un sujet de taille. À titre indicatif, sa description est aussi volumineuse que celle du langage lui-même dans la norme C++. Mais ce n'est pas tout, il faut impérativement avoir compris en profondeur les fonctionnalités les plus avancées du C++ pour appréhender correctement la bibliothèque standard. En particulier, tous les algorithmes et toutes les classes fournies par la bibliothèque sont susceptibles de travailler sur des données de type arbitraire. La bibliothèque utilise donc complètement la notion de `template`, et se base sur plusieurs abstractions des données manipulées et de leurs types afin de rendre générique l'implémentation des fonctionnalités. De plus, la bibliothèque utilise le mécanisme des exceptions afin de signaler les erreurs qui peuvent se produire lors de l'exécution des méthodes de ses classes et de ses fonctions. Enfin, un certain nombre de notions algorithmiques avancées sont utilisées dans toute la bibliothèque. La présentation qui sera faite sera donc progressive, tout en essayant de conserver un ordre logique. Tout comme pour la partie précédente, il est probable que plusieurs lectures seront nécessaires aux débutants pour assimiler toutes les subtilités de la bibliothèque.

Le premier chapitre de cette partie (Chapitre 13) présente les notions de base qui sont utilisées dans toute la libraire : encapsulation des fonctions de la bibliothèque C classique, classes de traits pour les types de base, notion d'itérateurs, de foncteurs, d'allocateurs mémoire et de complexité algorithmique. Le Chapitre 14 présente les types complémentaires que la bibliothèque standard C++ définit pour faciliter la vie du programmeur. Le plus important de ces types est sans doute la classe de gestion des chaînes de caractères basic_string. Le Chapitre 15 présente les notions de flux d'entrée / sortie standards, et la notion de tampon pour ces flux. Les mécanismes de localisation (c'est-à-dire les fonctions de paramétrage du programme en fonction des conventions et des préférences nationales) seront décrits dans le Chapitre 16. Le Chapitre 17 est sans doute l'un des plus importants, puisqu'il présente tous les conteneurs fournis par la bibliothèque standard. Enfin, le Chapitre 18 décrit les principaux algorithmes de la bibliothèque, qui permettent de manipuler les données stockées dans les conteneurs.

Les informations décrites ici sont basées sur la norme ISO 14882 du langage C++, et non sur la réalité des environnements C++ actuels. Il est donc fortement probable que bon nombre d'exemples fournis ici ne soient pas utilisables tels quels sur les environnements de développement existants sur le marché, bien que l'on commence à voir apparaître des environnements presque totalement respectueux de la norme maintenant. De légères différences dans l'interface des classes décrites peuvent également apparaître et nécessiter la modification de ces exemples. Cependant, à terme, tous les environnements de développement respecteront les interfaces spécifiées par la norme, et les programmes utilisant la bibliothèque standard seront réellement portables au niveau source.

Chapitre 13. Services et notions de base de la bibliothèque standard

La bibliothèque standard C++ fournit un certain nombre de fonctionnalités de base sur lesquelles toutes les autres fonctionnalités de la bibliothèque s'appuient. Ces fonctionnalités apparaissent comme des classes d'encapsulation de la bibliothèque C et des classes d'abstraction des principales constructions du langage. Ces dernières utilisent des notions très évoluées pour permettre une encapsulation réellement générique des types de base. D'autre part, la bibliothèque standard utilise la notion de complexité algorithmique pour définir les contraintes de performance des opérations réalisables sur ses structures de données ainsi que sur ses algorithmes. Bien que complexes, toutes ces notions sont omniprésentes dans toute la bibliothèque, aussi est-il extrêmement important de les comprendre en détail. Ce chapitre a pour but de vous les présenter et de les éclaircir.

13.1. Encapsulation de la bibliothèque C standard

La bibliothèque C définit un grand nombre de fonctions C standards, que la bibliothèque standard C++ reprend à son compte et complète par toutes ses fonctionnalités avancées. Pour bénéficier de ces fonctions, il suffit simplement d'inclure les fichiers d'en-tête de la bibliothèque C, tout comme on le faisait avec les programmes C classiques.

Toutefois, les fonctions ainsi déclarées par ces en-têtes apparaissent dans l'espace de nommage global, ce qui risque de provoquer des conflits de noms avec des fonctions homonymes (rappelons que les fonctions C ne sont pas surchargeables). Par conséquent, et dans un souci d'homogénéité avec le reste des fonctionnalités de la bibliothèque C++, un jeu d'en-têtes complémentaires a été défini pour les fonctions de la bibliothèque C. Ces en-têtes définissent tous leurs symboles dans l'espace de nommage `std::`, qui est réservé pour la bibliothèque standard C++.

Ces en-têtes se distinguent des fichiers d'en-tête de la bibliothèque C par le fait qu'ils ne portent pas d'extension `.h` et par le fait que leur nom est préfixé par la lettre '`c`'. Les en-têtes utilisables ainsi sont donc les suivants :

```
cassert
cctype
cerrno
cfloat
ciso646
climits
clocale
cmath
csetjmp
csignal
cstdarg
cstddef
cstdio
cstdlib
cstring
ctime
cwchar
cwctype
```

Par exemple, on peut réécrire notre tout premier programme que l'on a fait à la Section 1.2 de la manière suivante :

```
#include <cstdio>
```

```
long double x, y;

int main(void)
{
    std::printf("Calcul de moyenne\n");
    std::printf("Entrez le premier nombre : ");
    std::scanf("%Lf", &x);
    std::printf("\nEntrez le deuxième nombre : ");
    std::scanf("%Lf", &y);
    std::printf("\nLa valeur moyenne de %Lf et de %Lf est %Lf.\n",
        x, y, (x+y)/2);
    return 0;
}
```

Note : L'utilisation systématique du préfixe `std::` peut être énervante sur les grands programmes. On aura donc intérêt soit à utiliser les fichiers d'en-tête classiques de la bibliothèque C, soit à inclure une directive `using namespace std;` pour intégrer les fonctionnalités de la bibliothèque standard dans l'espace de nommage global.

Remarquez que la norme ne suppose pas que ces en-têtes soient des fichiers physiques. Les déclarations qu'ils sont supposés faire peuvent donc être réalisées à la volée par les outils de développement, et vous ne les trouverez pas forcément sur votre disque dur.

Certaines fonctionnalités fournies par la bibliothèque C ont été encapsulées dans des fonctionnalités équivalentes de la bibliothèque standard C++. C'est notamment le cas pour la gestion des locales et la gestion de certains types de données complexes. C'est également le cas pour la détermination des limites de représentation que les types de base peuvent avoir. Classiquement, ces limites sont définies par des macros dans les en-têtes de la bibliothèque C, mais elles sont également accessibles au travers de la classe `template` numeric_limits, définie dans l'en-tête `limits` :

```
// Types d'arrondis pour les flottants :
enum float_round_style
{
    round_indeterminate       = -1,
    round_toward_zero         =  0,
    round_to_nearest          =  1,
    round_toward_infinity     = 2,
    round_toward_neg_infinity = 3
};

template <class T>
class numeric_limits
{
public:
    static const bool is_specialized = false;
    static T min() throw();
    static T max() throw();
    static const int digits   = 0;
    static const int digits10 = 0;
    static const bool is_signed  = false;
    static const bool is_integer = false;
    static const bool is_exact   = false;
    static const int radix = 0;
    static T epsilon() throw();
```

```
    static T round_error() throw();
    static const int min_exponent   = 0;
    static const int min_exponent10 = 0;
    static const int max_exponent   = 0;
    static const int max_exponent10 = 0;
    static const bool has_infinity  = false;
    static const bool has_quiet_NaN = false;
    static const bool has_signaling_NaN = false;
    static const bool has_denorm        = false;
    static const bool has_denorm_loss   = false;
    static T infinity() throw();
    static T quiet_NaN() throw();
    static T signaling_NaN() throw();
    static T denorm_min() throw();
    static const bool is_iec559  = false;
    static const bool is_bounded = false;
    static const bool is_modulo  = false;
    static const bool traps      = false;
    static const bool tinyness_before = false;
    static const float_round_style
        round_style = round_toward_zero;
};
```

Cette classe `template` ne sert à rien en soi. En fait, elle est spécialisée pour tous les types de base du langage, et ce sont ces spécialisations qui sont réellement utilisées. Elles permettent d'obtenir toutes les informations pour chaque type grâce à leurs données membres et à leurs méthodes statiques.

Exemple 13-1. Détermination des limites d'un type

```
#include <iostream>
#include <limits>

using namespace std;

int main(void)
{
    cout << numeric_limits<int>::min() << endl;
    cout << numeric_limits<int>::max() << endl;
    cout << numeric_limits<int>::digits << endl;
    cout << numeric_limits<int>::digits10 << endl;
    return 0;
}
```

Ce programme d'exemple détermine le plus petit et le plus grand nombre représentable avec le type entier int, ainsi que le nombre de bits utilisés pour coder les chiffres et le nombre maximal de chiffres que les nombres en base 10 peuvent avoir en étant sûr de pouvoir être stockés tels quels.

13.2. Définition des exceptions standards

La bibliothèque standard utilise le mécanisme des exceptions du langage pour signaler les erreurs qui peuvent se produire à l'exécution au sein de ses fonctions. Elle définit pour cela un certain nombre de classes d'exceptions standards, que toutes les fonctionnalités de la bibliothèque sont susceptibles d'utiliser. Ces classes peuvent être utilisées telles quelles ou servir de classes de base à des classes d'exceptions personnalisées pour vos propres développements.

Ces classes d'exception sont presque toutes déclarées dans l'en-tête `stdexcept`, et dérivent de la classe de base exception. Cette dernière n'est pas déclarée dans le même en-tête et n'est pas utilisée directement, mais fournit les mécanismes de base de toutes les exceptions de la bibliothèque standard. Elle est déclarée comme suit dans l'en-tête `exception` :

```
class exception
{
public:
    exception() throw();
    exception(const exception &) throw();
    exception &operator=(const exception &) throw();
    virtual ~exception() throw();
    virtual const char *what() const throw();
};
```

Outre les constructeurs, opérateurs d'affectation et destructeurs classiques, cette classe définit une méthode `what` qui retourne une chaîne de caractères statique. Le contenu de cette chaîne de caractères n'est pas normalisé. Cependant, il sert généralement à décrire la nature de l'erreur qui s'est produite. C'est une méthode virtuelle, car elle est bien entendu destinée à être redéfinie par les classes d'exception spécialisées pour les différents types d'erreurs. Notez que toutes les méthodes de la classe exception sont déclarées comme ne pouvant pas lancer d'exceptions elle-mêmes, ce qui est naturel puisque l'on est déjà en train de traiter une exception lorsqu'on manipule des objets de cette classe.

L'en-tête `exception` contient également la déclaration de la classe d'exception bad_exception. Cette classe n'est, elle aussi, pas utilisée en temps normal. Le seul cas où elle peut être lancée est dans le traitement de la fonction de traitement d'erreur qui est appelée par la fonction `std::unexpected` lorsqu'une exception a provoqué la sortie d'une fonction qui n'avait pas le droit de la lancer. La classe bad_exception est déclarée comme suit dans l'en-tête `exception` :

```
class bad_exception : public exception
{
public:
    bad_exception() throw();
    bad_exception(const bad_exception &) throw();
    bad_exception &operator=(const bad_exception &) throw();
    virtual ~bad_exception() throw();
    virtual const char *what() const throw();
};
```

Notez que l'exception bad_alloc lancée par les gestionnaires de mémoire lorsque l'opérateur `new` ou l'opérateur `new[]` n'a pas réussi à faire une allocation n'est pas déclarée dans l'en-tête `stdexcept` non plus. Sa déclaration a été placée avec celle des opérateurs d'allocation mémoire, dans l'en-tête `new`. Cette classe dérive toutefois de la classe exception, comme le montre sa déclaration :

```
class bad_alloc : public exception
{
public:
    bad_alloc() throw();
    bad_alloc(const bad_alloc &) throw();
    bad_alloc &operator=(const bad_alloc &) throw();
    virtual ~bad_alloc() throw();
    virtual const char *what() const throw();
};
```

Les autres exceptions sont classées en deux grandes catégories. La première catégorie regroupe toutes les exceptions dont l'apparition traduit sans doute une erreur de programmation dans le programme, car elles ne devraient jamais se produire à l'exécution. Il s'agit des exceptions dites « d'erreurs dans la logique du programme » et, en tant que telles, dérivent de la classe d'exception logic_error. Cette classe est déclarée comme suit dans l'en-tête `stdexcept` :

```
class logic_error : public exception
{
public:
    logic_error(const string &what_arg);
};
```

Elle ne contient qu'un constructeur, permettant de définir la chaîne de caractères qui sera renvoyée par la méthode virtuelle `what`. Ce constructeur prend en paramètre cette chaîne de caractères sous la forme d'un objet de la classe string. Cette classe est définie par la bibliothèque standard afin de faciliter la manipulation des chaînes de caractères et sera décrite plus en détail dans la Section 14.1.

Les classes d'exception qui dérivent de la classe logic_error disposent également d'un constructeur similaire. Ces classes sont les suivantes :

- la classe domain_error, qui spécifie qu'une fonction a été appelée avec des paramètres sur lesquels elle n'est pas définie. Il faut contrôler les valeurs des paramètres utilisées lors de l'appel de la fonction qui a lancé cette exception ;
- la classe invalid_argument, qui spécifie qu'un des arguments d'une méthode ou d'une fonction n'est pas valide. Cette erreur arrive lorsqu'on utilise des valeurs de paramètres qui n'entrent pas dans le cadre de fonctionnement normal de la méthode appelée ; cela traduit souvent une mauvaise utilisation de la fonctionnalité correspondante ;
- la classe length_error, qui indique qu'un dépassement de capacité maximale d'un objet a été réalisé. Ces dépassements se produisent dans les programmes bogués, qui essaient d'utiliser une fonctionnalité au delà des limites qui avaient été fixées pour elle ;
- la classe out_of_range, qui spécifie qu'une valeur située en dehors de la plage de valeurs autorisées a été utilisée. Ce type d'erreur signifie souvent que les paramètres utilisés pour un appel de fonction ne sont pas corrects ou pas initialisés, et qu'il faut vérifier leur validité.

La deuxième catégorie d'exceptions correspond aux erreurs qui ne peuvent pas toujours être corrigées lors de l'écriture du programme, et qui font donc partie des événements naturels qui se produisent lors de son exécution. Elles caractérisent les erreurs d'exécution, et dérivent de la classe d'exception runtime_error. Cette classe est déclarée de la manière suivante dans l'en-tête `stdexcept` :

```
class runtime_error : public exception
{
public:
    runtime_error(const string &what_arg);
};
```

Elle s'utilise exactement comme la classe logic_error.

Les exceptions de la catégorie des erreurs d'exécution sont les suivantes :

- la classe range_error, qui signifie qu'une valeur est sortie de la plage de valeurs dans laquelle elle devait se trouver suite à un débordement interne à la bibliothèque ;

- la classe overflow_error, qui signifie qu'un débordement par valeurs supérieures s'est produit dans un calcul interne à la bibliothèque ;
- la classe underflow_error, qui signifie qu'un débordement par valeurs inférieures s'est produit dans un calcul interne à la bibliothèque.

13.3. Abstraction des types de données : les traits

Un certain nombre de classes ou d'algorithmes peuvent manipuler des types ayant une signification particulière. Par exemple, la classe string, que nous verrons plus loin, manipule des objets de type caractère. En réalité, ces classes et ces algorithmes peuvent travailler avec n'importe quels types pourvu que tous ces types se comportent de la même manière. La bibliothèque standard C++ utilise donc la notion de « *traits* », qui permet de définir les caractéristiques de ces types. Les traits sont définis dans des classes prévues à cet usage. Les classes et les algorithmes standards n'utilisent que les classes de traits pour manipuler les objets, garantissant ainsi une abstraction totale vis-à-vis de leurs types. Ainsi, il suffit de coder une spécialisation de la classe des traits pour un type particulier afin de permettre son utilisation dans les algorithmes génériques. La bibliothèque standard définit bien entendu des spécialisations pour les types de base du langage.

Par exemple, la classe de définition des traits des types de caractères est la classe template char_traits. Elle contient les définitions des types suivants :

- le type char_type, qui est le type représentant les caractères eux-mêmes ;
- le type int_type, qui est un type capable de contenir toutes les valeurs possibles pour les caractères, y compris la valeur spéciale du marqueur de fin de fichier ;
- le type off_type, qui est le type permettant de représenter les déplacements dans une séquence de caractères, ainsi que les positions absolues dans cette séquence. Ce type est signé car les déplacements peuvent être réalisés aussi bien vers le début de la séquence que vers la fin ;
- le type pos_type, qui est un sous-type du type off_type, et qui n'est utilisé que pour les déplacements dans les fonctions de positionnement des flux de la bibliothèque standard ;
- le type state_type, qui permet de représenter l'état courant d'une séquence de caractères dans les fonctions de conversion. Ce type est utilisé dans les fonctions de transcodage des séquences de caractères d'un encodage vers un autre.

> **Note :** Pour comprendre l'utilité de ce dernier type, il faut savoir qu'il existe plusieurs manières de coder les caractères. La plupart des méthodes utilisent un encodage à *taille fixe*, où chaque caractère est représenté par une valeur entière et une seule. Cette technique est très pratique pour les jeux de caractères contenant moins de 256 caractères, pour lesquels un seul octet est utilisé par caractère. Elle est également utilisée pour les jeux de caractères de moins de 65536 caractères, car l'utilisation de 16 bits par caractères est encore raisonable. En revanche, les caractères des jeux de caractères orientaux sont codés avec des valeurs numériques supérieures à 65536 par les encodages standards (Unicode et ISO 10646), et ne peuvent donc pas être stockés dans les types char ou wchar_t. Pour ces jeux de caractères, on utilise donc souvent des encodages à *taille variable*, où chaque caractère peut être représenté par un ou plusieurs octets selon sa nature et éventuellement selon sa position dans la chaîne de caractères.
>
> Pour ces encodages à taille variable, il est évident que le positionnement dans les séquences de caractères se fait en fonction du contexte de la chaîne, à savoir en fonction de la position du caractère précédent et parfois en fonction des caractères déjà analysés. Les algorithmes de

la bibliothèque standard qui manipulent les séquences de caractères doivent donc stocker le contexte courant lors de l'analyse de ces séquences. Elles le font grâce au type state_type de la classe des traits de ces caractères.

L'exemple suivant vous permettra de vérifier que le type char_type de la classe de définition des traits pour le type char est bien entendu le type char lui-même :

```
#include <iostream>
#include <typeinfo>
#include <string>

using namespace std;

int main(void)
{
    // Récupère les informations de typage des traits :
    const type_info &ti_trait =
        typeid(char_traits<char>::char_type);
    // Récupère les informations de typage directement :
    const type_info &ti_char = typeid(char);
    // Compare les types :
    cout << "Le nom du type caractère des traits est : " <<
        ti_trait.name() << endl;
    cout << "Le nom du type char est : " <<
        ti_char.name() << endl;
    if (ti_trait == ti_char)
        cout << "Les deux types sont identiques." << endl;
    else
        cout << "Ce n'est pas le même type." << endl;
    return 0;
}
```

La classe char_traits définit également un certain nombre de méthodes travaillant sur les types de caractères et permettant de réaliser les opérations de base sur ces caractères. Ces méthodes permettent essentiellement de comparer, de copier, de déplacer et de rechercher des caractères dans des séquences de caractères, en tenant compte de toutes les caractéristiques de ces caractères. Elle contient également la définition de la valeur spéciale utilisée dans les séquences de caractères pour marquer les fin de flux (« `EOF` », abréviation de l'anglais « End Of File »).

Par exemple, le programme suivant permet d'afficher la valeur utilisée pour spécifier une fin de fichier dans une séquence de caractères de type wchar_t :

```
#include <iostream>
#include <string>

using namespace std;

int main(void)
{
    char_traits<wchar_t>::int_type wchar_eof =
        char_traits<wchar_t>::eof();
    cout << "La valeur de fin de fichier pour wchar_t est : "
        << wchar_eof << endl;
```

```
    return 0;
}
```

Les autres méthodes de la classe de définition des traits des caractères, ainsi que les classes de définition des traits des autre types, ne seront pas décrites plus en détail ici. Elles sont essentiellement utilisées au sein des algorithmes de la bibliothèque standard et n'ont donc qu'un intérêt limité pour les programmeurs, mais il est important de savoir qu'elles existent.

13.4. Abstraction des pointeurs : les itérateurs

La bibliothèque standard définit un certain nombre de structures de données évoluées, qui permettent de stocker et de manipuler les objets utilisateur de manière optimale, évitant ainsi au programmeur d'avoir à réinventer la roue. On appelle ces structures de données des *conteneurs*. Ces conteneurs peuvent être manipulés au travers de fonctions spéciales, selon un grand nombre d'algorithmes possibles dont la bibliothèque dispose en standard. L'ensemble des fonctionnalités fournies par la bibliothèque permet de subvenir au besoin des programmeurs dans la majorité des cas. Nous détaillerons la notion de conteneur et les algorithmes disponibles plus loin dans ce document.

La manière d'accéder aux données des conteneurs dépend bien entendu de leur nature et de leur structure. Cela signifie qu'en théorie, il est nécessaire de spécialiser les fonctions permettant d'appliquer les algorithmes pour chaque type de conteneur existant. Cette technique n'est ni pratique, ni extensible, puisque les algorithmes fournis par la bibliothèque ne pourraient dans ce cas pas travailler sur des conteneurs écrits par le programmeur. C'est pour cette raison que la bibliothèque standard utilise une autre technique pour accéder aux données des conteneurs. Cette technique est basée sur la notion d'*itérateur*.

13.4.1. Notions de base et définition

Un itérateur n'est rien d'autre qu'un objet permettant d'accéder à tous les objets d'un conteneur donné, souvent séquentiellement, selon une interface standardisée. La dénomination d'itérateur provient donc du fait que les itérateurs permettent d'*itérer* sur les objets d'un conteneur, c'est-à-dire d'en parcourir le contenu en passant par tous ses objets.

Comme les itérateurs sont des objets permettant d'accéder à d'autres objets, ils ne représentent pas eux-mêmes ces objets, mais plutôt le moyen de les atteindre. Ils sont donc comparables aux pointeurs, dont ils ont exactement la même sémantique. En fait, les concepteurs de la bibliothèque standard se sont basés sur cette propriété pour définir l'interface des itérateurs, qui sont donc une extension de la notion de pointeur. Par exemple, il est possible d'écrire des expressions telles que « `*i` » ou « `++i` » avec un itérateur `i`. Tous les algorithmes de la bibliothèque, qui travaillent normalement sur des itérateurs, sont donc susceptibles de fonctionner avec des pointeurs classiques.

Bien entendu, pour la plupart des conteneurs, les itérateurs ne sont pas de simples pointeurs, mais des objets qui se comportent comme des pointeurs et qui sont spécifiques à chaque conteneur. Ainsi, les algorithmes sont écrits de manière uniforme, et ce sont les conteneurs qui fournissent les itérateurs qui leur sont appropriés afin de permettre l'accès à leurs données.

Il n'y a que trois manières d'obtenir un itérateur. Les itérateurs qui sont effectivement des pointeurs peuvent être obtenus naturellement en prenant l'adresse de l'élément auquel ils donnent accès. Les pointeurs ne doivent être utilisés en tant qu'itérateurs que pour accéder aux données d'un tableau, car la sémantique de l'arithmétique des pointeurs suppose que les éléments référencés successivement par un pointeur sont stockés en des emplacements contigus de la mémoire. Pour les itérateurs de conte-

neurs en revanche, il faut impérativement utiliser des méthodes spécifiques du conteneur pour obtenir des itérateurs. La plupart des conteneurs fournissent une méthode pour obtenir un itérateur initial, qui référence le premier élément du conteneur, et une méthode pour obtenir la valeur de l'itérateur lorsque le parcours du conteneur est achevé. Enfin, certains algorithmes et certaines méthodes des conteneurs peuvent retourner un itérateur à l'issu de leur traitement.

Quelle que soit la manière d'obtenir les itérateurs, leur validité est soumise à des limites. Premièrement, ils deviennent obligatoirement invalides dès lors que le conteneur auquel ils permettent d'accéder est détruit. De plus, les conteneurs gèrent leur structure de données de manière dynamique, et sont susceptibles de la réorganiser dès qu'on les manipule. On veillera donc à ne plus utiliser les itérateurs d'un conteneur dès qu'une méthode permettant de le modifier aura été appelée. Ne pas respecter cette règle conduirait, dans le meilleur des cas, à ne pas parcourir complètement l'ensemble des objets du conteneur, et dans le pire des cas, à planter immédiatement le programme.

13.4.2. Classification des itérateurs

La bibliothèque définit plusieurs catégories d'itérateurs qui contiennent des itérateurs plus ou moins puissants. Le comportement des itérateurs les plus puissants se rapproche beaucoup des pointeurs classiques, et quasiment toutes les opérations applicables aux pointeurs peuvent l'être à ces itérateurs. En revanche, les itérateurs des classes plus restrictives ne définissent qu'un sous-ensemble des opérations que les pointeurs supportent, et ne peuvent donc être utilisés que dans le cadre de ce jeu d'opérations réduit.

Les algorithmes de la bibliothèque n'utilisent que les itérateurs des classes les plus faibles permettant de réaliser leur travail. Ils s'imposent ces restrictions afin de garantir leur utilisation correcte même avec les itérateurs les plus simples. Bien entendu, comme les pointeurs disposent de toutes les fonctionnalités définies par les itérateurs, même les plus puissants, les algorithmes standards fonctionnent également avec des pointeurs. Autrement dit, la bibliothèque standard est écrite de façon à n'utiliser qu'une partie des opérations applicables aux pointeurs, afin de garantir que ce qui fonctionne avec des itérateurs fonctionne avec des pointeurs.

Les itérateurs de chaque catégorie possèdent toutes les propriétés des itérateurs des catégories inférieures. Il existe donc une hiérarchie dans la classification des itérateurs. Les catégories définies par la bibliothèque standard sont les suivantes :

- les itérateurs de la catégorie « *Output* » sont utilisés pour effectuer des affectations de valeurs aux données qu'ils référencent. Ces itérateurs ne peuvent donc être déréférencés par l'opérateur '$*$' que dans le cadre d'une affectation. Il est impossible de lire la valeur d'un itérateur de type Output, et on ne doit écrire dans la valeur qu'ils référencent qu'une fois au plus. Les algorithmes qui utilisent ces itérateurs doivent donc impérativement ne faire qu'une seule passe sur les données itérées ;
- les itérateurs de la catégorie « *Input* » sont similaires aux itérateurs de type Output, à ceci près qu'ils ne peuvent être déréférencés que pour lire une valeur. Contrairement aux itérateurs de type Output, il est possible de comparer deux itérateurs. Cependant, le fait que deux itérateurs soient égaux ne signifie aucunement que leurs successeurs le seront encore. Les algorithmes qui utilisent les itérateurs de type Input ne peuvent donc faire aucune hypothèse sur l'ordre de parcours utilisé par l'itérateur. Ce sont donc nécessairement des algorithmes en une passe ;
- les itérateurs de la catégorie « *Forward* » possèdent toutes les fonctionnalités des itérateurs de type Input et de type Output. Comme ceux-ci, ils ne peuvent passer que d'une valeur à la suivante, et jamais reculer ou revenir à une valeur déjà itérée. Les algorithmes qui utilisent des itérateurs de cette catégorie s'imposent donc de ne parcourir les données des conteneurs que dans un seul sens. Cependant, la restriction imposée sur l'égalité des opérateurs de type Input est levée, ce qui signifie que plusieurs parcours successifs se feront dans le même ordre. Les algorithmes peuvent

effectuer plusieurs parcours, par exemple en copiant la valeur initiale de l'itérateur et en parcourant le conteneur plusieurs fois avec chaque copie ;

- les itérateurs de la catégorie « *Bidirectionnal* » disposent de toutes les fonctionnalités des itérateurs de type Forward, mais lèvent la restriction sur le sens de parcours. Ces itérateurs peuvent donc revenir sur les données déjà itérées, et les algorithmes qui les utilisent peuvent donc travailler en plusieurs passes, dans les deux directions ;
- enfin, les itérateurs de la catégorie « *RandomAccess* » sont les plus puissants. Ils fournissent toutes les fonctionnalités des itérateurs de type Bidirectionnal, plus la possibilité d'accéder aux éléments des conteneurs par l'intermédiaire d'un index en un temps constant. Il n'y a donc plus de notion de sens de parcours, et les données peuvent être accédées comme les données d'un tableau. Il est également possible d'effectuer les opérations classiques de l'arithmétique des pointeurs sur ces itérateurs.

Tous les itérateurs de la bibliothèque standard dérivent de la classe de base suivante :

```
template <class Category,
    class T, class Distance = ptrdiff_t,
    class Pointer = T*, class Reference = T &>
struct iterator
{
    typedef T         value_type;
    typedef Distance  difference_type;
    typedef Pointer   pointer;
    typedef Reference reference;
    typedef Category  iterator_category;
};
```

Cette classe est déclarée dans l'en-tête `iterator`.

Cette classe définit les types de base des itérateurs, à savoir : le type des valeurs référencées, le type de la différence entre deux itérateurs dans les calculs d'arithmétique des pointeurs, le type des pointeurs des valeurs référencées par l'itérateur, le type des références pour ces valeurs et la catégorie de l'itérateur. Ce dernier type doit être l'une des classes suivantes, également définies par la bibliothèque standard :

- input_iterator_tag, pour les itérateurs de la catégorie des itérateurs de type Input ;
- output_iterator_tag, pour les itérateurs de la catégorie des itérateurs de type Output ;
- forward_iterator_tag, pour les itérateurs de la catégorie des itérateurs de type Forward ;
- bidirectionnal_iterator_tag, pour les itérateurs de la catégorie des itérateurs bidirectionnels ;
- random_access_iterator_tag, pour les itérateurs de la catégorie des itérateurs à accès complet.

Notez que le type par défaut pour la différence entre deux pointeurs est le type ptrdiff_t, qui est utilisé classiquement pour les pointeurs normaux. De même, le type pointeur et le type référence correspondent respectivement, par défaut, aux types T * et T &. Pour les itérateurs pour lesquels ces types n'ont pas de sens, le type utilisé est void, ce qui permet de provoquer une erreur de compilation si on cherche à les utiliser.

Ces types sont utilisés par les itérateurs nativement, cependant, ils ne le sont généralement pas par les algorithmes. En effet, ceux-ci sont susceptibles d'être appelées avec des pointeurs normaux, et les pointeurs ne définissent pas tous ces types. C'est pour cette raison qu'une classe de traits a été définie

pour les itérateurs par la bibliothèque standard. Cette classe est déclarée comme suit dans l'en-tête `iterator` :

```
template <class Iterator>
struct iterator_traits
{
    typedef Iterator::value_type        value_type;
    typedef Iterator::difference_type   difference_type;
    typedef Iterator::pointer           pointer;
    typedef Iterator::reference         reference;
    typedef Iterator::iterator_category iterator_category;
};
```

La classe des traits permet donc d'obtenir de manière indépendante de la nature de l'itérateur la valeur des types fondamentaux de l'itérateur. Comme ces types n'existent pas pour les pointeurs classiques, cette classe est spécialisée de la manière suivante :

```
template <class T>
struct iterator_traits<T *>
{
    typedef T         value_type;
    typedef ptrdiff_t difference_type;
    typedef T         *pointer;
    typedef T         &reference;
    typedef random_access_iterator_tag iterator_category;
};
```

Ainsi, le type iterator_traits<itérateur>::difference_type renverra toujours le type permettant de stocker la différence entre deux itérateurs, que ceux-ci soient des itérateurs ou des pointeurs normaux.

Pour comprendre l'importance des traits des itérateurs, prenons l'exemple de deux fonctions fournies par la bibliothèque standard permettant d'avancer un itérateur d'un certain nombre d'étapes, et de calculer la différence entre deux itérateurs. Il s'agit respectivement des fonctions `advance` et `distance`. Ces fonctions devant pouvoir travailler avec n'importe quel itérateur, et n'importe quel type de donnée pour exprimer la différence entre deux itérateurs, elles utilisent la classe des traits. Elles sont déclarées de la manière suivante dans l'en-tête `iterator` :

```
template <class InputIterator, class Distance>
void advance(InputIterator &i, Distance n);

template <class InputIterator>
iterator_traits<InputIterator>::difference_type
    distance(InputIterator first, InputIterator last);
```

Notez que le type de retour de la fonction `distance` est Iterator::difference_type pour les itérateurs normaux, et ptrdiff_t pour les pointeurs.

> **Note :** Ces deux méthodes ne sont pas très efficaces avec les itérateurs de type Forward, car elles doivent parcourir les valeurs de ces itérateurs une à une. Cependant, elles sont spécialisées pour les itérateurs de type plus évolués (en particulier les itérateurs à accès complet), et sont donc plus efficaces pour eux. Elles permettent donc de manipuler les itérateurs de manière uniforme, sans pour autant compromettre les performances.

13.4.3. Itérateurs adaptateurs

Les itérateurs sont une notion extrêmement utilisée dans toute la bibliothèque standard, car ils regroupent toutes les fonctionnalités permettant d'effectuer un traitement séquentiel des données. Cependant, il n'existe pas toujours d'itérateur pour les sources de données que l'on manipule. La bibliothèque standard fournit donc ce que l'on appelle des itérateurs *adaptateurs*, qui permettent de manipuler ces structures de données en utilisant la notion d'itérateur même si ces structures ne gèrent pas elles-mêmes la notion d'itérateur.

13.4.3.1. Adaptateurs pour les flux d'entrée / sortie standards

Les flux d'entrée / sortie standards de la bibliothèque sont normalement utilisés avec les opérations '>>' et '<<', respectivement pour recevoir et pour envoyer des données. Il n'existe pas d'itérateur de type Input et de type Output permettant de lire et d'écrire sur ces flux. La bibliothèque définit donc des adaptateurs permettant de construire ces itérateurs.

L'itérateur adaptateur pour les flux d'entrée est implémenté par la classe `template` istream_iterator. Cet adaptateur est déclaré comme suit dans l'en-tête `iterator` :

```
template <class T, class charT, class traits = char_traits<charT>,
    class Distance=ptrdiff_t>
class istream_iterator :
    public iterator<input_iterator_tag, T, Distance,
        const T *, const T &>
{
public:
    typedef charT  char_type;
    typedef traits trait_type;
    typedef basic_istream<char, traits> istream_type;
    istream_iterator();
    istream_iterator(istream_iterator &flux);
    istream_iterator(const istream_iterator<T, charT, traits,
        Distance> &flux);
    ~istream_iterator();
    const T &operator*() const;
    const T *operator->() const;
    istream_iterator<T, charT, traits, Distance> &operator++();
    istream_iterator<T, charT, traits, Distance> operator++(int);
};
```

Les opérateurs d'égalité et d'inégalité sont également définis pour cet itérateur.

Comme vous pouvez le constater d'après cette déclaration, il est possible de construire un itérateur sur un flux d'entrée permettant de lire les données de ce flux une à une. S'il n'y a plus de données à lire sur ce flux, l'itérateur prend la valeur de l'itérateur de fin de fichier pour le flux. Cette valeur est celle qui est attribuée à tout nouvel itérateur construit sans flux d'entrée. L'exemple suivant présente comment faire la somme de plusieurs nombres lus sur le flux d'entrée, et de l'afficher lorsqu'il n'y a plus de données à lire.

Exemple 13-2. Itérateurs de flux d'entrée

```
#include <iostream>
#include <iterator>

using namespace std;
```

```
int main(void)
{
    double somme = 0;
    istream_iterator<double, char> is(cin);
    while (is != istream_iterator<double, char>())
    {
        somme = somme + *is;
        ++is;
    }
    cout << "La somme des valeurs lue est : " <<
        somme << endl;
    return 0;
}
```

Vous pourrez essayer ce programme en tapant plusieurs nombres successivement puis en envoyant un caractère de fin de fichier avec la combinaison de touches `CTRL + Z`. Ce caractère provoquera la sortie de la boucle `while` et affichera le résultat.

L'itérateur adaptateur pour les flux de sortie fonctionne de manière encore plus simple, car il n'y a pas à faire de test sur la fin de fichier. Il est déclaré comme suit dans l'en-tête `iterator` :

```
template <class T, class charT = char, class traits = char_traits<charT> >
class ostream_iterator :
    public iterator<output_iterator_tag, void, void, void, void>
{
public:
    typedef charT  char_type;
    typedef traits trait_type;
    typedef basic_ostream<charT, traits> ostream_type;
    ostream_iterator(ostream_type &flux);
    ostream_iterator(ostream_type &flux, const charT *separateur);
    ostream_iterator(const ostream_iterator<T, charT, traits> &flux);
    ~ostream_iterator();
    ostream_iterator<T, charT, traits> &operator=(const T &valeur);
    ostream_iterator<T, charT, traits> &operator*();
    ostream_iterator<T, charT, traits> &operator++();
    ostream_iterator<T, charT, traits> &operator++(int);
};
```

Cet itérateur est de type Output, et ne peut donc être déréférencé que dans le but de faire une écriture dans l'objet ainsi obtenu. Ce déréférencement retourne en fait l'itérateur lui-même, si bien que l'écriture provoque l'appel de l'opérateur d'affectation de l'itérateur. Cet opérateur envoie simplement les données sur le flux de sortie que l'itérateur prend en charge et renvoie sa propre valeur afin de réaliser une nouvelle écriture. Notez que les opérateurs d'incrémentation existent également, mais ne font strictement rien. Ils ne sont là que pour permettre d'utiliser ces itérateurs comme de simples pointeurs.

L'itérateur ostream_iterator peut envoyer sur le flux de sortie un texte intercalaire entre chaque donnée qu'on y écrit. Ce texte peut servir à insérer des séparateurs entre les données. Cette fonctionnalité peut s'avérer très pratique pour l'écriture de données formatées. Le texte à insérer automatiquement doit être passé en tant que deuxième argument du constructeur de l'itérateur.

Exemple 13-3. Itérateur de flux de sortie

```
#include <iostream>
#include <iterator>

using namespace std;

const char *texte[6] = {
    "Bonjour", "tout", "le", "monde", "!", NULL
};

int main(void)
{
    ostream_iterator<const char *, char> os(cout, " ");
    int i = 0;
    while (texte[i] != NULL)
    {
        *os = texte[i];  // Le déréférencement est facultatif.
        ++os;            // Cette ligne est facultative.
        ++i;
    }
    cout << endl;
    return 0;
}
```

Il existe également des adaptateurs pour les tampons de flux d'entrée / sortie basic_streambuf. Le premier adaptateur est implémenté par la classe `template` istreambuf_iterator. Il permet de lire les données provenant d'un tampon de flux basic_streambuf aussi simplement qu'en manipulant un pointeur et en lisant la valeur de l'objet pointé. Le deuxième adaptateur, ostreambuf_iterator, permet quant à lui d'écrire dans un tampon en affectant une nouvelle valeur à l'objet référencé par l'itérateur. Ces adaptateurs fonctionnent donc exactement de la même manière que les itérateurs pour les flux d'entrée / sortie formatés. En particulier, la valeur de fin de fichier que prend l'itérateur d'entrée peut être récupérée à l'aide du constructeur par défaut de la classe istreambuf_iterator, instanciée pour le type de tampon utilisé.

> **Note :** L'opérateur de d'incrémentation suffixé des itérateurs istreambuf_iterator a un type de retour particulier qui permet de représenter la valeur précédente de l'itérateur avant incrémentation. Les objets de ce type sont toujours déréférençables à l'aide de l'opérateur '`*`'. La raison de cette particularité est que le contenu du tampon peut être modifié après l'appel de l'opérateur '`operator ++(int)`', mais l'ancienne valeur de cet itérateur doit toujours permettre d'accéder à l'objet qu'il référençait. La valeur retournée par l'itérateur contient donc une sauvegarde de cet objet et peut se voir appliquer l'opérateur de déréférencement '`*`' par l'appelant afin d'en récupérer la valeur.
>
> La notion de tampon de flux sera présentée en détail dans la Section 15.2.

13.4.3.2. Adaptateurs pour l'insertion d'éléments dans les conteneurs

Les itérateurs fournis par les conteneurs permettent d'en parcourir le contenu et d'obtenir une référence sur chacun de leurs éléments. Ce comportement est tout à fait classique et constitue même une des bases de la notion d'itérateur. Toutefois, l'insertion de nouveaux éléments dans un conteneur ne peut se faire que par l'intermédiaire des méthodes spécifiques aux conteneurs. La bibliothèque standard C++ définit donc des adaptateurs pour des itérateurs dits d'*insertion*, qui permettent d'insérer

des éléments dans des conteneurs par un simple déréférencement et une écriture. Grâce à ces adaptateurs, l'insertion des éléments dans les conteneurs peut être réalisée de manière uniforme, de la même manière qu'on écrirait dans un tableau qui se redimensionnerait automatiquement, à chaque écriture.

Il est possible d'insérer les nouveaux éléments en plusieurs endroits dans les conteneurs. Ainsi, les éléments peuvent être placés au début du conteneur, à sa fin, ou après un élément donné. Bien entendu, ces notions n'ont de sens que pour les conteneurs qui ne sont pas ordonnés, puisque dans le cas contraire, la position de l'élément inséré est déterminée par le conteneur lui-même.

La classe `template` back_insert_iterator est la classe de l'adaptateur d'insertion en fin de conteneur. Elle est déclarée comme suit dans l'en-tête `iterator` :

```
template <class Container>
class back_insert_iterator :
    public iterator<output_iterator_tag, void, void, void, void>
{
public:
    typedef Container container_type;
    explicit back_insert_iterator(Container &conteneur);
    back_insert_iterator<Container> &
        operator=(const typename Container::value_type &valeur);
    back_insert_iterator<Container> &operator*();
    back_insert_iterator<Container> &operator++();
    back_insert_iterator<Container> operator++(int);
};
```

Comme vous pouvez le constater, les objets des instances cette classe peuvent être utilisés comme des itérateurs de type Output. L'opérateur de déréférencement '`*`' renvoie l'itérateur lui-même, si bien que les affectations sur les itérateurs déréférencés sont traitées par l'opérateur '`operator=`' de l'itérateur lui-même. C'est donc cet opérateur qui ajoute l'élément à affecter à la fin du conteneur auquel l'itérateur permet d'accéder, en utilisant la méthode `push_back` de ce dernier.

De même, la classe `template` front_insert_iterator de l'adaptateur d'insertion en tête de conteneur est déclarée comme suit dans l'en-tête `iterator` :

```
template <class Container>
class front_insert_iterator :
    public iterator<output_iterator_tag, void, void, void, void>
{
public:
    typedef Container container_type;
    explicit front_insert_iterator(Container &conteneur);
    front_insert_iterator<Container> &
        operator=(const typename Container::value_type &valeur);
    front_insert_iterator<Container> &operator*();
    front_insert_iterator<Container> &operator++();
    front_insert_iterator<Container> operator++(int);
};
```

Son fonctionnement est identique à celui de back_insert_iterator, à ceci près qu'il effectue les insertions des éléments au début du conteneur, par l'intermédiaire de sa méthode `push_front`.

Enfin, la classe `template` de l'adaptateur d'itérateur d'insertion à une position donnée est déclarée comme suit :

```
template <class Container>
class insert_iterator :
    public iterator<output_iterator_tag, void, void, void, void>
```

```
{
public:
    typedef Container container_type;
    insert_iterator(Container &conteneur,
        typename Container::iterator position);
    insert_iterator<Container> &
        operator=(const typename Container::value_type &valeur);
    insert_iterator<Container> &operator*();
    insert_iterator<Container> &operator++();
    insert_iterator<Container> operator++(int);
};
```

Le constructeur de cette classe prend en paramètre, en plus du conteneur sur lequel l'itérateur d'insertion doit travailler, un itérateur spécifiant la position à laquelle les éléments doivent être insérés. Les éléments sont insérés juste avant l'élément référencé par l'itérateur fourni en paramètre. De plus, ils sont insérés séquentiellement, les uns après les autres, dans leur ordre d'affectation via l'itérateur.

La bibliothèque standard C++ fournit trois fonctions `template` qui permettent d'obtenir les trois types d'itérateur d'insertion pour chaque conteneur. Ces fonctions sont déclarées comme suit dans l'en-tête `iterator` :

```
template <class Container>
back_insert_iterator<Container>
    back_inserter(Container &conteneur);

template <class Container>
front_insert_iterator<Container>
    front_inserter(Container &conteneur);

template <class Container, class Iterator>
insert_iterator<Container>
    inserter(Container &conteneur, Iterator position);
```

Le programme suivant utilise un itérateur d'insertion pour remplir une liste d'élément, avant d'en afficher le contenu.

Exemple 13-4. Itérateur d'insertion

```
#include <iostream>
#include <list>
#include <iterator>

using namespace std;

// Définit le type liste d'entier :
typedef list<int> li_t;

int main()
{
    // Crée une liste :
    li_t lst;
    // Insère deux éléments dans la liste de la manière classique :
    lst.push_back(1);
    lst.push_back(10);
```

```
    // Récupère un itérateur référençant le premier élément :
    li_t::iterator it = lst.begin();
    // Passe au deuxième élément :
    ++it;
    // Construit un itérateur d'insertion pour insérer de nouveaux
    // éléments avant le deuxième élément de la liste :
    insert_iterator<li_t> ins_it = inserter(lst, it);
    // Insère les éléments avec cet itérateur :
    for (int i = 2; i < 10; ++i)
    {
        *ins_it = i;
        ++ins_it;
    }
    // Affiche le contenu de la liste :
    it = lst.begin();
    while (it != lst.end())
    {
        cout << *it << endl;
        ++it;
    }
    return 0;
}
```

La manière d'utiliser le conteneur de type list sera décrite en détail dans le Chapitre 17.

13.4.3.3. Itérateur inverse pour les itérateurs bidirectionnels

Les itérateurs bidirectionnels et les itérateurs à accès aléatoire peuvent être parcourus dans les deux sens. Pour ces itérateurs, il est donc possible de définir un itérateur associé dont le sens de parcours est inversé. Le premier élément de cet itérateur est donc le dernier élément de l'itérateur associé, et inversement.

La bibliothèque standard C++ définit un adaptateur permettant d'obtenir un itérateur inverse facilement dans l'en-tête `iterator`. Il s'agit de la classe `template` reverse_iterator :

```
template <class Iterator>
class reverse_iterator :
    public iterator<
        iterator_traits<Iterator>::iterator_category,
        iterator_traits<Iterator>::value_type,
        iterator_traits<Iterator>::difference_type,
        iterator_traits<Iterator>::pointer,
        iterator_traits<Iterator>::reference>
{
public:
    typedef Iterator iterator_type;
    reverse_iterator();
    explicit reverse_iterator(Iterator iterateur);
    Iterator base() const;
    Reference operator*() const;
    Pointer operator->() const;
    reverse_iterator &operator++();
    reverse_iterator operator++(int);
    reverse_iterator &operator--();
    reverse_iterator operator--(int);
    reverse_iterator operator+(Distance delta) const;
```

```
    reverse_iterator &operator+=(Distance delta);
    reverse_iterator operator-(Distance delta) const;
    reverse_iterator &operator-=(Distance delta);
    Reference operator[](Distance delta) const;
};
```

Les opérateurs de comparaison classiques et d'arithmétique des pointeurs externes `operator+` et `operator-` sont également définis dans cet en-tête.

Le constructeur de cet adaptateur prend en paramètre l'itérateur associé dans le sens inverse duquel le parcours doit se faire. L'itérateur inverse pointera alors automatiquement sur l'élément précédent l'élément pointé par l'itérateur passé en paramètre. Ainsi, si on initialise l'itérateur inverse avec la valeur de fin de l'itérateur direct, il référencera le dernier élément que l'itérateur direct aurait référencé avant d'obtenir sa valeur finale dans un parcours des éléments du conteneur. La valeur de fin de l'itérateur inverse peut être obtenue en construisant un itérateur inverse à partir de la valeur de début de l'itérateur direct.

> **Note :** Notez que le principe spécifiant que l'adresse suivant celle du dernier élément d'un tableau doit toujours être une adresse valide est également en vigueur pour les itérateurs. La valeur de fin d'un itérateur est assimilable à cette adresse, pointant sur l'emplacement suivant le dernier élément d'un tableau, et n'est pas plus déréférençable, car elle se trouve en dehors du tableau. Cependant, elle peut être utilisée dans les calculs d'arithmétique des pointeurs, et c'est exactement ce que fait l'adaptateur reverse_iterator.

La méthode `base` permet de récupérer la valeur de l'itérateur direct associé à l'itérateur inverse. On prendra garde que l'itérateur renvoyé par cette méthode ne référence pas le même élément que celui référencé par l'itérateur inverse. En effet, l'élément référencé est toujours l'élément suivant l'élément référencé par l'itérateur inverse, en raison de la manière dont cet itérateur est initialisé. Par exemple, l'itérateur inverse référence le dernier élément du conteneur lorsqu'il vient d'être intialisé avec la valeur de fin de l'itérateur directe, valeur qui représente le dernier élément passé. De même, lorsque l'itérateur inverse a pour valeur sa valeur de fin d'itération (ce qui représente l'élément précédent le premier élément du conteneur en quelque sorte), l'itérateur direct référence le premier élément du conteneur.

En fait, les itérateurs inverses sont utilisés en interne par les conteneurs pour fournir des itérateurs permettant de parcourir leurs données dans le sens inverse. Le programmeur n'aura donc généralement pas besoin de construire des itérateurs inverses lui-même, il utilisera plutôt les itérateurs fournis par les conteneurs.

Exemple 13-5. Utilisation d'un itérateur inverse

```
#include <iostream>
#include <list>
#include <iterator>

using namespace std;

// Définit le type liste d'entier :
typedef list<int> li_t;

int main(void)
{
    // Crée une nouvelle liste :
    li_t li;
```

```
    // Remplit la liste :
    for (int i = 0; i < 10; ++i)
        li.push_back(i);
    // Affiche le contenu de la liste à l'envers :
    li_t::reverse_iterator rev_it = li.rbegin();
    while (rev_it != li.rend())
    {
        cout << *rev_it << endl;
        ++rev_it;
    }
    return 0;
}
```

13.5. Abstraction des fonctions : les foncteurs

La plupart des algorithmes de la bibliothèque standard, ainsi que quelques méthodes des classes qu'elle fournit, donnent la possibilité à l'utilisateur d'appliquer une fonction aux données manipulées. Ces fonctions peuvent être utilisées pour différentes tâches, comme pour comparer deux objets par exemple, ou tout simplement pour en modifier la valeur.

Cependant, la bibliothèque standard n'utilise pas ces fonctions directement, mais a plutôt recours à une abstraction des fonctions : les *foncteurs*. Un *foncteur* n'est rien d'autre qu'un objet dont la classe définit l'opérateur fonctionnel '`()`'. Les foncteurs ont la particularité de pouvoir être utilisés exactement comme des fonctions puisqu'il est possible de leur appliquer leur opérateur fonctionnel selon une écriture similaire à un appel de fonction. Cependant, ils sont un peu plus puissants que de simples fonctions, car ils permettent de transporter, en plus du code de l'opérateur fonctionnel, des paramètres additionnels dans leurs données membres. Les foncteurs constituent donc une fonctionnalité extrêmement puissante qui peut être très pratique en de nombreux endroits. En fait, comme on le verra plus loin, toute fonction peut être transformée en foncteur. Les algorithmes de la bibliothèque standard peuvent donc également être utilisés avec des fonctions classiques moyennant cette petite transformation.

Les algorithmes de la bibliothèque standard qui utilisent des foncteurs sont déclarés avec un paramètre `template` dont la valeur sera celle du foncteur permettant de réaliser l'opération à appliquer sur les données en cours de traitement. Au sein de ces algorithmes, les foncteurs sont utilisés comme de simples fonctions, et la bibliothèque standard ne fait donc pas d'autre hypothèse sur leur nature. Cependant, il est nécessaire de ne donner que des foncteurs en paramètres aux algorithmes de la bibliothèque standard, pas des fonctions. C'est pour cette raison que la bibliothèque standard définit un certain nombre de foncteurs standards afin de faciliter la tâche du programmeur.

13.5.1. Foncteurs prédéfinis

La bibliothèque n'utilise, dans ses algorithmes, que des foncteurs qui ne prennent qu'un ou deux paramètres. Les foncteurs qui prennent un paramètre et un seul sont dits « *unaires* », alors que les foncteurs qui prennent deux paramètres sont qualifiés de « *binaires* ». Afin de faciliter la création de foncteurs utilisables avec ses algorithmes, la bibliothèque standard définit deux classes de base qui pour les foncteurs unaires et binaires. Ces classes de base sont les suivantes :

```
template <class Arg, class Result>
struct unary_function
```

```
{
    typedef Arg    argument_type;
    typedef Result result_type;
};

template <class Arg1, class Arg2, class Result>
struct binary_function
{
    typedef Arg1   first_argument_type;
    typedef Arg2   second_argument_type;
    typedef Result result_type;
};
```

Ces classes sont définies dans l'en-tête `functional`.

La bibliothèque définit également un certain nombre de foncteurs standards qui encapsulent les opérateurs du langage dans cet en-tête. Ces foncteurs sont les suivants :

```
template <class T>
struct plus : binary_function<T, T, T>
{
    T operator()(const T &operande1, const T &operande2) const;
};

template <class T>
struct minus : binary_function<T, T, T>
{
    T operator()(const T &operande1, const T &operande2) const;
};

template <class T>
struct multiplies : binary_function<T, T, T>
{
    T operator()(const T &operande1, const T &operande2) const;
};

template <class T>
struct divides : binary_function<T, T, T>
{
    T operator()(const T &operande1, const T &operande2) const;
};

template <class T>
struct modulus : binary_function<T, T, T>
{
    T operator()(const T &operande1, const T &operande2) const;
};

template <class T>
struct negate : unary_function<T, T>
{
    T operator()(const T &operande) const;
};

template <class T>
struct equal_to : binary_function<T, T, bool>
{
    bool operator()(const T &operande1, const T &operande2) const;
```

```
};

template <class T>
struct not_equal_to : binary_function<T, T, bool>
{
    bool operator()(const T &operande1, const T &operande2) const;
};

template <class T>
struct greater : binary_function<T, T, bool>
{
    bool operator()(const T &operande1, const T &operande2) const;
};

template <class T>
struct less : binary_function<T, T, bool>
{
    bool operator()(const T &operande1, const T &operande2) const;
};

template <class T>
struct greater_equal : binary_function<T, T, bool>
{
    bool operator()(const T &operande1, const T &operande2) const;
};

template <class T>
struct less_equal : binary_function<T, T, bool>
{
    bool operator()(const T &operande1, const T &operande2) const;
};
```

Ces foncteurs permettent d'utiliser les principaux opérateurs du langage comme des fonctions classiques dans les algorithmes de la bibliothèque standard.

Exemple 13-6. Utilisation des foncteurs prédéfinis

```
#include <iostream>
#include <functional>

using namespace std;

// Fonction template prenant en paramètre deux valeurs
// et un foncteur :
template <class T, class F>
T applique(T i, T j, F foncteur)
{
    // Applique l'opérateur fonctionnel au foncteur
    // avec comme arguments les deux premiers paramètres :
    return foncteur(i, j);
}

int main(void)
{
    // Construit le foncteur de somme :
    plus<int> foncteur_plus;
    // Utilise ce foncteur pour faire faire une addition
```

```
    // à la fonction "applique" :
    cout << applique(2, 3, foncteur_plus) << endl;
    return 0;
}
```

Dans l'exemple précédent, la fonction `template applique` prend en troisième paramètre un foncteur et l'utilise pour réaliser l'opération à faire avec les deux premiers paramètres. Cette fonction ne peut théoriquement être utilisée qu'avec des objets disposant d'un opérateur fonctionnel '`()`', et pas avec des fonctions normales. La bibliothèque standard fournit donc les adaptateurs suivants, qui permettent de convertir respectivement n'importe quelle fonction unaire ou binaire en foncteur :

```
template <class Arg, class Result>
class pointer_to_unary_function :
    public unary_function<Arg, Result>
{
public:
    explicit pointer_to_unary_function(Result (*fonction)(Arg));
    Result operator()(Arg argument1) const;
};

template <class Arg1, Arg2, Result>
class pointer_to_binary_function :
    public binary_function<Arg1, Arg2, Result>
{
public:
    explicit pointer_to_binary_function(Result (*fonction)(Arg1, Arg2));
    Result operator()(Arg1 argument1, Arg2 argument2) const;
};

template <class Arg, class Result>
pointer_to_unary_function<Arg, Result>
    ptr_fun(Result (*fonction)(Arg));

template <class Arg, class Result>
pointer_to_binary_function<Arg1, Arg2, Result>
    ptr_fun(Result (*fonction)(Arg1, Arg2));
```

Les deux surcharges de la fonction `template ptr_fun` permettent de faciliter la construction d'un foncteur unaire ou binaire à partir du pointeur d'une fonction du même type.

Exemple 13-7. Adaptateurs de fonctions

```
#include <iostream>
#include <functional>

using namespace std;

template <class T, class F>
T applique(T i, T j, F foncteur)
{
    return foncteur(i, j);
}

// Fonction classique effectuant une multiplication :
int mul(int i, int j)
{
    return i * j;
```

```
}

int main(void)
{
    // Utilise un adaptateur pour transformer le pointeur
    // sur la fonction mul en foncteur :
    cout << applique(2, 3, ptr_fun(&mul)) << endl;
    return 0;
}
```

Note : En réalité le langage C++ est capable d'appeler une fonction directement à partir de son adresse, sans déréférencement. De plus le nom d'une fonction représente toujours sont adresse, et est donc converti implicitement par le compilateur en pointeur de fonction. Par conséquent, il est tout à fait possible d'utiliser la fonction `template applique` avec une autre fonction à deux paramètres, comme dans l'appel suivant :

```
applique(2, 3, mul);
```

Cependant, cette écriture provoque la conversion implicite de l'identificateur `mul` en pointeur de fonction prenant deux entiers en paramètres et renvoyant un entier, d'une part, et l'appel de la fonction `mul` par l'intermédiaire de son pointeur sans déréférencement dans la fonction `template applique` d'autre part. Cette écriture est donc acceptée par le compilateur par tolérance, mais n'est pas rigoureusement exacte.

La bibliothèque standard C++ définit également des adaptateurs pour les pointeurs de méthodes non statiques de classes. Ces adaptateurs se construisent comme les adaptateurs de fonctions statiques classiques, à ceci près que leur constructeur prend un pointeur de méthode de classe et non un pointeur de fonction normale. Ils sont déclarés de la manière suivante dans l'en-tête `functional` :

```
template <class Result, class Class>
class mem_fun_t :
    public unary_function<Class *, Result>
{
public:
    explicit mem_fun_t(Result (Class::*methode)());
    Result operator()(Class *pObjet);
};

template <class Result, class Class, class Arg>
class mem_fun1_t :
    public binary_function<Class *, Arg, Result>
{
public:
    explicit mem_fun1_t(Result (Class::*methode)(Arg));
    Result operator()(Class *pObjet, Arg argument);
};

template <class Result, class Class>
class mem_fun_ref_t :
    public unary_function<Class, Result>
{
public:
    explicit mem_fun_ref_t(Result (Class::*methode)());
```

```
    Result operator()(Class &objet);
};

template <class Result, class Class, class Arg>
class mem_fun1_ref_t :
    public binary_function<Class, Arg, Result>
{
public:
    explicit mem_fun1_ref_t(Result (Class::*methode)(Arg));
    Result operator()(Class &objet, Arg argument);
};

template <class Result, class Class>
mem_fun_t<Result, Class> mem_fun(Result (Class::*methode)());

template <class Result, class Class, class Arg>
mem_fun1_t<Result, Class> mem_fun(Result (Class::*methode)(Arg));

template <class Result, class Class>
mem_fun_ref_t<Result, Class> mem_fun_ref(Result (Class::*methode)());

template <class Result, class Class, class Arg>
mem_fun1_ref_t<Result, Class>
    mem_fun_ref(Result (Class::*methode)(Arg));
```

Comme vous pouvez le constater d'après leurs déclarations les opérateurs fonctionnels de ces adaptateurs prennent en premier paramètre soit un pointeur sur l'objet sur lequel le foncteur doit travailler (adaptateurs mem_fun_t et mem_fun1_t), soit une référence sur cet objet (adaptateurs mem_fun_ref_t et mem_fun1_ref_t). Le premier paramètre de ces foncteurs est donc réservé pour l'objet sur lequel la méthode encapsulée doit être appelée.

En fait, la liste des adaptateurs présentée ci-dessus n'est pas exhaustive. En effet, chaque adaptateur présenté est doublé d'un autre adaptateur, capable de convertir les fonctions membres `const`. Il existe donc huit adaptateurs au total permettant de construire des foncteurs à partir des fonctions membres de classes. Pour diminuer cette complexité, la bibliothèque standard définit plusieurs surcharges pour les fonctions `mem_fun` et `mem_fun_ref`, qui permettent de construire tous ces foncteurs plus facilement, sans avoir à se soucier de la nature des pointeurs de fonction membre utilisés. Il est fortement recommandé de les utiliser plutôt que de chercher à construire ces objets manuellement.

13.5.2. Prédicats et foncteurs d'opérateurs logiques

Les foncteurs qui peuvent être utilisés dans une expression logique constituent une classe particulière : les *prédicats*. Un *prédicat* est un foncteur dont l'opérateur fonctionnel renvoie un booléen. Les prédicats ont donc un sens logique, et caractérisent une propriété qui ne peut être que vraie ou fausse.

La bibliothèque standard fournit des prédicats prédéfinis qui effectuent les opérations logiques des opérateurs logiques de base du langage. Ces prédicats sont également déclarés dans l'en-tête `functional` :

```
template <class T>
struct logical_and :
    binary_function<T, T, bool>
{
    bool operator()(const T &operande1, const T &operande2) const;
```

```
};

template <class T>
struct logical_or :
    binary_function<T, T, bool>
{
    bool operator()(const T &operande1, const T &operande2) const;
};
```

Ces foncteurs fonctionnent exactement comme les foncteurs vus dans la section précédente.

La bibliothèque standard définit aussi deux foncteurs particuliers, qui permettent d'effectuer la négation d'un autre prédicat. Ces deux foncteurs travaillent respectivement sur les prédicats unaires et sur les prédicats binaires :

```
template <class Predicate>
class unary_negate :
    public unary_function<typename Predicate::argument_type, bool>
{
public:
    explicit unary_negate(const Predicate &predicat);
    bool operator()(const argument_type &argument) const;
};

template <class Predicate>
class binary_negate :
    public binary_function<typename Predicate::first_argument_type,
        typename Predicate::second_argument_type, bool>
{
public:
    explicit binary_negate(const Predicate &predicat);
    bool operator()(const first_argument_type &argument1,
        const second_argument_type &argument2) const;
};

template <class Predicate>
unary_negate<Predicate> not1(const Predicate &predicat);

template <class Predicate>
binary_negate<Predicate> not2(const Predicate &predicat);
```

Les fonctions `not1` et `not2` servent à faciliter la construction d'un prédicat inverse pour les prédicats unaires et binaires.

13.5.3. Foncteurs réducteurs

Nous avons vu que la bibliothèque standard ne travaillait qu'avec des foncteurs prenant au plus deux arguments. Certains algorithmes n'utilisant que des foncteurs unaires, ils ne sont normalement pas capables de travailler avec les foncteurs binaires. Toutefois, si un des paramètres d'un foncteur binaire est fixé à une valeur donnée, celui-ci devient unaire, puisque seul le deuxième paramètre peut varier. Il est donc possible d'utiliser des foncteurs binaires même avec des algorithmes qui n'utilisent que des foncteurs unaires, à la condition de fixer l'un des paramètres.

La bibliothèque standard définit des foncteurs spéciaux qui permettent de transformer tout foncteur binaire en foncteur unaire à partir de la valeur de l'un des paramètres. Ces foncteurs effectuent une opération dite de *réduction* car ils réduisent le nombre de paramètres du foncteur binaire à un. Pour

cela, ils définissent un opérateur fonctionnel à un argument, qui applique l'opérateur fonctionnel du foncteur binaire à cet argument et à une valeur fixe qu'ils mémorisent en donnée membre.

Ces foncteurs réducteurs sont déclarés, comme les autres foncteurs, dans l'en-tête `fonctional` :

```
template <class Operation>
class binder1st :
    public unary_function<typename Operation::second_argument_type,
        typename Operation::result_type>
{
protected:
    Operation op;
    typename Operation::first_argument_type value;
public:
    binder1st(const Operation &foncteur,
        const typename Operation::first_argument_type &valeur);
    result_type operator()(const argument_type &variable) const;
};

template <class Operation>
class binder2nd :
    public unary_function<typename Operation::first_argument_type,
        typename Operation::result_type>
{
protected:
    Operation op;
    typename Operation::second_argument_type value;
public:
    binder2nd(const Operation &foncteur,
        const typename Operation::second_argument_type &valeur);
    result_type operator()(const argument_type &variable) const;
};

template <class Operation, class T>
binder1st<Operation> bind1st(const Operation &foncteur, const T &valeur);

template <class Operation, class T>
binder2nd<Operation> bind2nd(const Operation &foncteur, const T &valeur);
```

Il existe deux jeux de réducteurs, qui permettent de réduire les foncteurs binaires en fixant respectivement leur premier ou leur deuxième paramètre. Les réducteurs qui figent le premier paramètre peuvent être construits à l'aide de la fonction `template bind1st`, et ceux qui figent la valeur du deuxième paramètre peuvent l'être à l'aide de la fonction `bind2nd`.

Exemple 13-8. Réduction de foncteurs binaires

```
#include <iostream>
#include <functional>

using namespace std;

// Fonction template permettant d'appliquer une valeur
// à un foncteur unaire. Cette fonction ne peut pas
// être utilisée avec un foncteur binaire.
template <class Foncteur>
typename Foncteur::result_type applique(
```

```
    Foncteur f,
    typename Foncteur::argument_type valeur)
{
    return f(valeur);
}

int main(void)
{
    // Construit un foncteur binaire d'addition d'entiers :
    plus<int> plus_binaire;
    int i;
    for (i = 0; i < 10; ++i)
    {
        // Réduit le foncteur plus_binaire en fixant son
        // premier paramètre à 35. Le foncteur unaire obtenu
        // est ensuite utilisé avec la fonction applique :
        cout << applique(bind1st(plus_binaire, 35), i) << endl;
    }
    return 0;
}
```

13.6. Gestion personnalisée de la mémoire : les allocateurs

L'une des plus grandes forces de la bibliothèque standard est de donner aux programmeurs le contrôle total de la gestion de la mémoire pour leurs objets. En effet, les conteneurs peuvent être amenés à créer un grand nombre d'objets, dont le comportement peut être très différent selon leur type. Si, dans la majorité des cas, la gestion de la mémoire effectuée par la bibliothèque standard convient, il peut parfois être nécessaire de prendre en charge soi-même les allocations et les libérations de la mémoire pour certains objets.

La bibliothèque standard utilise pour cela la notion d'*allocateur*. Un allocateur est une classe C++ disposant de méthodes standards que les algorithmes de la bibliothèque peuvent appeler lorsqu'elles désirent allouer ou libérer de la mémoire. Pour cela, les conteneurs de la bibliothèque standard C++ prennent tous un paramètre `template` représentant le type des allocateurs mémoire qu'ils devront utiliser. Bien entendu, la bibliothèque standard fournit un allocateur par défaut, et ce paramètre `template` prend par défaut la valeur de cet allocateur. Ainsi, les programmes qui ne désirent pas spécifier un allocateur spécifique pourront simplement ignorer ce paramètre `template`.

Les autres programmes pourront définir leur propre allocateur. Cet allocateur devra évidemment fournir toutes les fonctionnalités de l'allocateur standard, et satisfaire à quelques contraintes particulières. L'interface des allocateurs est fournie par la déclaration de l'allocateur standard, dans l'en-tête `memory` :

```
template <class T>
class allocator
{
public:
    typedef size_t    size_type;
    typedef ptrdiff_t difference_type;
    typedef T         *pointer;
    typedef const T   *const_pointer;
```

```
    typedef T         &reference;
    typedef const T   &const_reference;
    typedef T         value_type;
    template <class U>
    struct rebind
    {
        typedef allocator<U> other;
    };

    allocator() throw();
    allocator(const allocator &) throw();
    template <class U>
    allocator(const allocator<U> &) throw();
    ~allocator() throw();
    pointer address(reference objet);
    const_pointer address(const_reference objet) const;
    pointer allocate(size_type nombre,
        typename allocator<void>::const_pointer indice);
    void deallocate(pointer adresse, size_type nombre);
    size_type max_size() const throw();
    void construct(pointer adresse, const T &valeur);
    void destroy(pointer adresse);
};

// Spécialisation pour le type void pour éliminer les références :
template <>
class allocator<void>
{
public:
    typedef void       *pointer;
    typedef const void *const_pointer;
    typedef void       value_type;
    template <class U>
    struct rebind
    {
        typedef allocator<U> other;
    };
};
```

Vous noterez que cet allocateur est spécialisé pour le type void, car certaines méthodes et certains `typedef` n'ont pas de sens pour ce type de donnée.

Le rôle de chacune des méthodes des allocateurs est très clair et n'appelle pas beaucoup de commentaires. Les deux surcharges de la méthode `address` permettent d'obtenir l'adresse d'un objet alloué par cet allocateur à partir d'une référence. Les méthodes `allocate` et `deallocate` permettent respectivement de réaliser une allocation de mémoire et la libération du bloc correspondant. La méthode `allocate` prend en paramètre le nombre d'objets qui devront être stockés dans le bloc à allouer et un pointeur fournissant des informations permettant de déterminer l'emplacement où l'allocation doit se faire de préférence. Ce dernier paramètre peut ne pas être pris en compte par l'implémentation de la bibliothèque standard que vous utilisez et, s'il l'est, son rôle n'est pas spécifié. Dans tous les cas, s'il n'est pas nul, ce pointeur doit être un pointeur sur un bloc déjà alloué par cet allocateur et non encore libéré. La plupart des implémentations chercheront à allouer un bloc adjacent à celui fourni en paramètre, mais ce n'est pas toujours le cas. De même, notez que le nombre d'objets spécifié à la méthode `deallocate` doit exactement être le même que celui utilisé pour l'allocation dans l'appel correspondant à `allocate`. Autrement dit, l'allocateur ne mémorise pas lui-même la taille des blocs mémoire qu'il a fourni.

Note : Le pointeur passé en paramètre à la méthode `allocate` n'est ni libéré, ni réalloué, ni réutilisé par l'allocateur. Il ne s'agit donc pas d'une modification de la taille mémoire du bloc fourni en paramètre, et ce bloc devra toujours être libéré indépendamment de celui qui sera alloué. Ce pointeur n'est utilisé par les implémentations que comme un indice fourni à l'allocateur afin d'optimiser les allocations de blocs dans les algorithmes et les conteneurs internes.

La méthode `allocate` peut lancer l'exception bad_alloc en cas de manque de mémoire ou si le nombre d'objets spécifié en paramètre est trop gros. Vous pourrez obtenir le nombre maximal que la méthode `allocate` est capable d'accepter grâce à la méthode `max_size` de l'allocateur.

Les deux méthodes `construct` et `destroy` permettent respectivement de construire un nouvel objet et d'en détruire un à l'adresse indiquée en paramètre. Elles doivent être utilisées lorsqu'on désire appeler le constructeur ou le destructeur d'un objet stocké dans une zone mémoire allouée par cet allocateur et non par les opérateurs `new` et `delete` du langage (rappelons que ces opérateurs effectuent ce travail automatiquement). Pour effectuer la construction d'un nouvel objet, `construct` utilise l'opérateur `new` avec placement, et pour le détruire, `destroy` appelle directement le destructeur de l'objet.

Note : Les méthodes `construct` et `destroy` n'effectuent pas l'allocation et la libération de la mémoire elles-mêmes. Ces opérations doivent être effectuées avec les méthodes `allocate` et `deallocate` de l'allocateur.

Exemple 13-9. Utilisation de l'allocateur standard

```
#include <iostream>
#include <memory>

using namespace std;

class A
{
public:
    A();
    A(const A &);
    ~A();
};

A::A()
{
    cout << "Constructeur de A" << endl;
}

A::A(const A &)
{
    cout << "Constructeur de copie de A" << endl;
}

A::~A()
{
    cout << "Destructeur de A" << endl;
}

int main(void)
```

```
{
    // Construit une instance de l'allocateur standard pour la classe A :
    allocator<A> A_alloc;

    // Alloue l'espace nécessaire pour stocker cinq instances de A :
    allocator<A>::pointer p = A_alloc.allocate(5);

    // Construit ces instances et les initialise :
    A init;
    int i;
    for (i=0; i<5; ++i)
        A_alloc.construct(p+i, init);
    // Détruit ces instances :
    for (i=0; i<5; ++i)
        A_alloc.destroy(p+i);

    // Reconstruit ces 5 instances :
    for (i=0; i<5; ++i)
        A_alloc.construct(p+i, init);
    // Destruction finale :
    for (i=0; i<5; ++i)
        A_alloc.destroy(p+i);

    // Libère la mémoire :
    A_alloc.deallocate(p, 5);
    return 0;
}
```

Vous voyez ici l'intérêt que peut avoir les allocateurs de la bibliothèque standard. Les algorithmes peuvent contrôler explicitement la construction et la destruction des objets, et surtout les dissocier des opérations d'allocation et de libération de la mémoire. Ainsi, un algorithme devant effectuer beaucoup d'allocations mémoire pourra, s'il le désire, effectuer ces allocations une bonne fois pour toutes grâce à l'allocateur standard, et n'effectuer les opérations de construction et de destruction des objets que lorsque cela est nécessaire. En procédant ainsi, le temps passé dans les routines de gestion de la mémoire est éliminé et l'algorithme est d'autant plus performant. Inversement, un utilisateur expérimenté pourra définir son propre allocateur mémoire adapté aux objets qu'il voudra stocker dans un conteneur. En imposant au conteneur de la bibliothèque standard d'utiliser cet allocateur personnalisé, il obtiendra des performances optimales.

La définition d'un allocateur maison consiste simplement à implémenter une classe `template` disposant des mêmes méthodes et types que ceux définis par l'allocateur allocator. Toutefois, il faut savoir que la bibliothèque impose des contraintes sur la sémantique de ces méthodes :

- toutes les instances de la classe `template` de l'allocateur permettent d'accéder à la même mémoire. Ces instances sont donc interchangeables et il est possible de passer de l'une à l'autre à l'aide de la structure `template` rebind et de son `typedef` other. Notez que le fait d'encapsuler ce `typedef` dans une structure `template` permet de simuler la définition d'un type `template` ;
- toutes les instances d'un allocateur d'un type donné permettent également d'accéder à la même mémoire. Cela signifie qu'il n'est pas nécessaire de disposer d'une instance globale pour chaque allocateur, il suffit simplement de créer un objet local d'une des instances de la classe `template` de l'allocateur pour allouer et libérer de la mémoire. Notez ici la différence avec la contrainte précédente : cette contrainte porte ici sur les objets instances des classes `template` instanciées, alors que la contrainte précédente portait sur les instances elles-mêmes de la classe `template` de l'allocateur ;

- toutes les méthodes de l'allocateur doivent s'exécuter dans un temps amorti constant (cela signifie que le temps d'exécution de ces méthodes est majoré par une borne supérieure fixe, qui ne dépend pas du nombre d'allocation déjà effectuées ni de la taille du bloc de mémoire demandé) ;
- les méthodes `allocate` et `deallocate` sont susceptibles d'utiliser les opérateurs `new` et `delete` du langage. Ce n'est pas une obligation, mais cette contrainte signifie que les programmes qui redéfinissent ces deux opérateurs doivent être capable de satisfaire les demandes de l'allocateur standard ;
- les types pointer, const_pointer, size_type et difference_type doivent être égaux respectivement aux types T *, const T*, size_t et ptrdiff_t. En fait, cette contrainte n'est imposée que pour les allocateurs destinés à être utilisés par les conteneurs de la bibliothèque standard, mais il est plus simple de la généraliser à tous les cas d'utilisation.

Pour terminer ce tour d'horizon des allocateurs, sachez que la bibliothèque standard définit également un type itérateur spécial permettant de stocker des objets dans une zone de mémoire non initialisée. Cet itérateur, nommé raw_storage_iterator, est de type Output et n'est utilisé qu'en interne par la bibliothèque standard. De même, la bibliothèque définit des algorithmes permettant d'effectuer des copies brutes de blocs mémoire et d'autres manipulations sur les blocs alloués par les allocateurs. Ces algorithmes sont également utilisés en interne, et ne seront donc pas décrits plus en détail ici.

13.7. Notion de complexité algorithmique

En aucun endroit la norme C++ ne spécifie la manière de réaliser une fonctionnalité. En effet, elle n'impose ni les structures de données, ni les algorithmes à utiliser. Les seules choses qui sont spécifiées par la norme sont les interfaces bien entendu (c'est-à-dire les noms des classes, des objets et les signatures des fonctions et des méthodes) et la sémantique des diverses opérations réalisables. Cependant, la norme C++ ne permet pas de réaliser toutes ces fonctionnalités n'importe comment, car elle impose également des contraintes de performances sur la plupart de ses algorithmes ainsi que sur les méthodes des conteneurs. Ces contraintes sont exprimées généralement en terme de complexité algorithmique, aussi est-il nécessaire de préciser un peu cette notion.

> **Note :** En pratique, les contraintes de complexité imposées par la bibliothèque standard sont tout simplement les plus fortes réalisables. En d'autres termes, on ne peut pas faire mieux que les algorithmes de la bibliothèque standard.

13.7.1. Généralités

La nature des choses veut que plus un programme a de données à traiter, plus il prend du temps pour le faire. Cependant, certains algorithmes se comportent mieux que d'autres lorsque le nombre des données à traiter augmente. Par exemple, un algorithme mal écrit peut voir son temps d'exécution croître exponentiellement avec la quantité de données à traiter, alors qu'un algorithme bien étudié aurait n'aurait été plus lent que proportionnellement à ce même nombre. En pratique, cela signifie que cet algorithme est tout simplement inutilisable lorsque le nombre de données augmente. Par exemple, le fait de doubler la taille de l'ensemble des données à traiter peut engendrer un temps de calcul quatre fois plus long, alors que le temps d'exécution de l'algorithme bien écrit n'aurait été que du double seulement. Et si le nombre de données est triplé et non doublé, cet algorithme demandera huit fois plus de temps, là où le triple seulement est nécessaire. Si l'on prend quatre fois plus de données, le temps

sera multiplié par soixante-quatre. On voit clairement que les choses ne vont pas en s'améliorant quand le nombre de données à traiter augmente...

En réalité, il est relativement rare de considérer le temps d'exécution pour qualifier les performances d'un algorithme. En effet, le calcul du temps d'exécution n'est pas toujours possible d'une part, parce qu'il se base sur des paramètres a priori inconnus, et n'est pas toujours ce qui est intéressant au niveau du coût d'autre part. Pour illustrer ce dernier point, supposons que chaque opération effectuée par l'algorithme coûte une certaine quantité d'énergie. Dans certains contextes, il est plus important de s'intéresser à l'énergie dépensée qu'au temps passé pour effectuer l'ensemble des traitements. Or certaines opérations peuvent prendre relativement peu de temps, mais coûter très cher énergétiquement parlant, et l'optimisation du temps d'exécution ne donne pas forcément la meilleure solution. Un autre exemple est tout simplement celui de la lecture de secteurs sur un disque dur. La lecture de ces secteurs en soi ne prend pas tellement de temps, en revanche le déplacement de la tête de lecture se fait en un temps d'accès considérablement plus grand. Les algorithmes de gestion des entrées / sorties sur disque des systèmes d'exploitation cherchent donc naturellement à diminuer au maximum ces déplacements en réorganisant en conséquence les requêtes de lecture et d'écriture.

Il est donc généralement beaucoup plus simple de compter le nombre d'opérations que les algorithmes effectuent lors de leur déroulement, car cette donnée est bien moins spécifique au contexte d'utilisation de l'algorithme. Bien entendu, toutes les opérations effectuées par un algorithme n'ont pas le même coût dans un contexte donné, et de plus ce coût varie d'un contexte d'utilisation à un autre. La complexité d'un algorithme doit donc toujours s'exprimer en nombre d'opérations élémentaires d'un certain type, étant entendu que les opérations de ce type sont celles qui coûtent le plus cher selon les critères choisis...

Remarquez que les opérations qui sont réalisées par un algorithme peuvent être elles-mêmes relativement complexes. Par exemple, un algorithme qui applique une fonction sur chaque donnée à traiter peut utiliser une fonction inimaginablement complexe. Cependant, cela ne nous intéresse pas dans la détermination de la complexité de cet algorithme. Bien entendu, ce qu'il faut compter, c'est le nombre de fois que cette fonction est appelée, et la complexité de l'algorithme doit se calculer indépendamment de celle de cette fonction. L'opération élémentaire de l'algorithme est donc ici tout simplement l'appel de cette fonction, aussi complexe soit-elle.

13.7.2. Notions mathématiques de base et définition

Le nombre des opérations élémentaires effectuées par un algorithme est une fonction directe du nombre de données à traiter. La complexité d'un algorithme est donc directement reliée à cette fonction : plus elle croît rapidement avec le nombre de données à traiter, plus la complexité de l'algorithme est grande.

En réalité, la fonction exacte donnant le nombre d'opérations élémentaires effectuées par un algorithme n'est pas toujours facile à calculer. Cependant, il existe toujours une fonction plus simple qui dispose du même comportement que la fonction du nombre d'opérations de l'algorithme quand le nombre de données à traiter augmente. Cette « forme simplifiée » n'est en fait rien d'autre que la partie croissant le plus vite avec le nombre de données, car lorsque celui-ci tend vers l'infini, c'est elle qui devient prédominante. Cela signifie que si l'on trace le graphe de la fonction, sa forme finit par ressembler à celle de sa forme simplifiée lorsque le nombre de données à traiter devient grand.

La formulation complète de la fonction du nombre d'opérations réalisées par un algorithme n'importe donc pas tant que cela, ce qui est intéressant, c'est sa forme simplifiée. En effet, non seulement elle est plus simple (à exprimer, à manipuler et bien évidemment à retenir), mais en plus elle caractérise correctement le comportement de l'algorithme sur les grands nombres. La complexité d'un algorithme est donc, par définition, le terme prépondérant dans la fonction donnant le nombre d'opérations élémentaires effectuées par l'algorithme en fonction du nombre des données à traiter.

Mathématiquement parlant, le fait que la forme simplifiée d'une fonction se comporte comme la fonction elle-même à l'infini se traduit simplement en disant que les termes d'ordre inférieurs sont écrasés par le terme de premier ordre. Par conséquent, si l'on divise une fonction par l'autre, les termes d'ordre inférieur deviennent négligeables et la valeur du rapport tend à se stabiliser vers les grand nombres. Autrement dit, il est possible de trouver deux constantes `A` et `B` positives et non nulles telles que, à partir d'une certaine valeur de `n`, la triple inéquation $0 \leq A \times c(n) \leq f(n) \leq B \times c(n)$, dans laquelle `c(n)` est la forme simplifiée de la fonction `f(n)`, est toujours vérifiée. La fonction `f(n)` est donc, en quelque sortes, encadrée par deux « gendarmes » qui suivent le même « trajet » : celui de la fonction `c(n)`.

> **Note :** Notez que cette formulation n'utilise pas le rapport des fonctions `f(n)` et `c(n)` directement. Elle est donc toujours valide, même lorsque ces deux fonctions sont nulles, ce qui aurait posé des problèmes si l'on avait utilisé un rapport.

En fait, la limite inférieure $A \times c(n)$ ne nous intéresse pas spécialement. En effet, seul le coût maximal d'un algorithme est intéressant, car s'il coûte moins cher que prévu, personne ne s'en plaindra... Il est donc courant d'utiliser une formulation plus simple et plus connue des mathématiciens, dans laquelle seule la dernière inéquation est utilisée. On dit alors que la fonction `f(n)` est en grand O de `c(n)` (ce qui se note « `O(c(n))` »). Cela signifie qu'il existe une constante `A` telle que, pour toutes les valeurs de `n` supérieures à une valeur suffisamment grande, la double inéquation $0 \leq f(n) \leq A \times c(n)$ est toujours vérifiée.

> **Note :** La notion de grand O permet donc de donner une borne supérieure de la complexité de la fonction. En fait, si `f(n)` est en `O(c(n))`, elle l'est pour toutes les fonctions plus grandes que `c(n)`. Toutefois, en général, on cherche à déterminer la plus petite fonction `c(n)` qui est un grand O de `f(n)`.
>
> Il est évident que si une fonction `f(n)` dispose d'une forme simplifiée `c(n)`, elle est en O(c(n)). En effet, l'inéquation supérieure est toujours vérifiée, on ne fait ici qu'ignorer la deuxième inéquation de la définition de la forme simplifiée.

13.7.3. Interprétation pratique de la complexité

Toutes ces notions peuvent vous paraître assez abstraites, mais il est important de bien comprendre ce qu'elles signifient. Il est donc peut-être nécessaire de donner quelques exemples de complexité parmi celles que l'on rencontre le plus couramment.

Tout d'abord, une complexité de `1` pour un algorithme signifie tout simplement que son coût d'exécution est constant, quel que soit le nombre de données à traiter. Notez bien ici que l'on parle de coût d'exécution et non de durée. Le coût est ici le nombre d'opérations élémentaires effectuées par cet algorithme. Les algorithmes de complexité `1` sont évidemment les plus intéressants, mais ils sont hélas assez rares ou tout simplement triviaux.

Généralement, les algorithmes ont une complexité de `n`, leur coût d'exécution est donc proportionnel au nombre de données à traiter. C'est encore une limite acceptable, et généralement acceptée comme une conséquence « logique » de l'augmentation du nombre de données à traiter. Certains algorithmes sont en revanche nettement moins performants et ont une complexité en n^2, soit le carré du nombre des éléments à traiter. Cette fois, cela signifie que leur coût d'exécution a tendance à croître très rapidement lorsqu'il y a de plus en plus de données. Par exemple, si l'on double le nombre de données, le coût d'exécution de l'algorithme ne double pas, mais quadruple. Et si l'on triple le nombre de

données, ce coût devient neuf fois plus grand. Ne croyez pas pour autant que les algorithmes de ce type soient rares ou mauvais. On ne peut pas toujours, hélas, faire autrement...

Il existe même des algorithmes encore plus coûteux, qui utilisent des exposants bien supérieurs à 2. Inversement, certains algorithmes extrêmement astucieux permettent de réduire les complexités `n` ou n^2 en `ln(n)` ou `n×ln(n)`, ils sont donc nettement plus efficaces.

> **Note :** La fonction `ln(n)` est la fonction logarithmique, qui est la fonction inverse de l'exponentielle, bien connue pour sa croissance démesurée. La fonction logarithme évolue beaucoup moins vite que son argument, en l'occurrence `n` dans notre cas, et a donc tendance à « écraser » le coût des algorithmes qui l'ont pour complexité.

Enfin, pour terminer ces quelques notions de complexité algorithmique, sachez que l'on peut évaluer la difficulté d'un problème à partir de la complexité du meilleur algorithme qui permet de le résoudre. Par exemple, il a été démontré que le tri d'un ensemble de `n` éléments ne peut pas se faire en mieux que `n×ln(n)` opérations (et on sait le faire, ce qui est sans doute le plus intéressant de l'affaire). Malheureusement, il n'est pas toujours facile de déterminer la complexité d'un problème. Il existe même toute une classe de problèmes extrêmement difficiles à résoudre pour lesquels on ne sait même pas si leur solution optimale est polynomiale ou non. En fait, on ne sait les résoudre qu'avec des algorithmes de complexité exponentielle (si vous ne savez pas ce que cela signifie, en un mot, cela veut dire que c'est une véritable catastrophe). Cependant, cela ne veut pas forcément dire qu'on ne peut pas faire mieux, mais tout simplement qu'on n'a pas pu trouver une meilleure solution, ni même prouver qu'il y en avait une ! Toutefois, tous ces problèmes sont liés et, si on trouve une solution polynomiale pour l'un d'entre eux, on saura résoudre aussi facilement tous ses petits camarades. Ces problèmes appartiennent tous à la classe des problèmes dits « NP-complets ».

Chapitre 14. Les types complémentaires

Le C++ étant un langage basé sur le C, il souffre des mêmes limitations concernant les types de données avancés que celui-ci. Pour pallier cet inconvénient, la bibliothèque standard C++ définit des types complémentaires sous forme de classes C++, éventuellement `template`, et permettant de satisfaire aux besoins les plus courants. Parmi ces types, on notera avant tout le type basic_string, qui permet de manipuler les chaînes de caractères de manière plus simple et plus sûre qu'avec des pointeurs et des tableaux de caractères. Mais la bibliothèque standard définit également des classes utilitaires qui permettent de manipuler les autres types plus facilement, ainsi que des types capables d'utiliser toutes les ressources de la machine pour les calculs numériques avancés.

14.1. Les chaînes de caractères

La classe `template` basic_string de la bibliothèque standard, déclarée dans l'en-tête `string`, facilite le travail du programmeur et permet d'écrire du code manipulant des textes de manière beaucoup plus sûre. En effet, cette classe encapsule les chaînes de caractères C classiques et fournissent des services extrêmement intéressants qui n'étaient pas disponibles auparavant. En particulier, la classe basic_string dispose des caractéristiques suivantes :

- compatibilité quasi-totale avec les chaînes de caractères C standards ;
- gestion des chaînes à taille variable ;
- prise en charge de l'allocation dynamique de la mémoire et de sa libération en fonction des besoins et de la taille des chaînes manipulées ;
- définition des opérateurs de concaténation, de comparaison et des principales méthodes de recherche dans les chaînes de caractères ;
- intégration totale dans la bibliothèque standard, en particulier au niveau des flux d'entrée / sortie.

Comme il l'a été dit plus haut, la classe basic_string est une classe `template`. Cela signifie qu'elle est capable de prendre en charge des chaînes de n'importe quel type de caractère. Pour cela, elle ne se base que sur la classe des traits du type de caractère manipulé. Il est donc parfaitement possible de l'utiliser avec des types définis par l'utilisateur, pourvu que la classe des traits des caractères soit définie pour ces types. Bien entendu, la classe basic_string peut être utilisée avec les types de caractères du langage, à savoir char et wchar_t.

Les déclarations de l'en-tête `string` sont essentiellement les suivantes :

```
template <class charT, class traits = char_traits<charT>,
    class Allocator = allocator<charT> >
class basic_string
{
public:
    // Types
    typedef traits                    traits_type;
    typedef typename traits::char_type value_type;
    typedef Allocator                 allocator_type;
    typedef typename Allocator::size_type       size_type;
    typedef typename Allocator::difference_type difference_type;
    typedef typename Allocator::reference       reference_type;
    typedef typename Allocator::const_reference const_reference;
```

```
typedef typename Allocator::pointer         pointer;
typedef typename Allocator::const_pointer   const_pointer;

// Constante utilisée en interne et représentant la valeur maximale
// du type size_type :
static const size_type npos = static_cast<size_type>(-1);

// Constructeurs et destructeur :
explicit basic_string(const Allocator &allocateur = Allocator());
basic_string(const basic_string &source, size_type debut = 0,
    size_type longueur = npos, const Allocator &allocateur = Allocator());
basic_string(const charT *chaine, size_type nombre,
    const Allocator &allocateur = Allocator());
basic_string(const charT *chaine,
    const Allocator &allocateur = Allocator());
basic_string(size_type nombre, charT caractere,
    const Allocator &allocateur  = Allocator());
template <class InputIterator>
basic_string(InputIterator debut, InputIterator fin,
    const Allocator &allocateur  = Allocator());
~basic_string();

// Itérateurs :
typedef type_privé iterator;
typedef type_privé const iterator;
typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
iterator       begin();
const_iterator begin() const;
iterator       end();
const_iterator end() const;
reverse_iterator       rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator       rend();
const_reverse_iterator rend() const;

// Accesseurs :
size_type size() const;
size_type length() const;
size_type max_size() const;
size_type capacity() const;
bool empty() const;
allocator_type get_allocator() const;

// Manipulateurs :
void resize(size_type taille, charT caractere);
void resize(size_type taille);
void reserve(size_type taille = 0);

// Accès aux données de la chaîne :
const_reference operator[](size_type index) const;
reference operator[](size_type index);
const_reference at(size_type index) const;
reference at(size_type index);
const charT *c_str() const;
const charT *data() const;
size_type copy(charT *destination, size_type taille,
```

```
    size_type debut = 0) const;
basic_string substr(size_type debut = 0, size_type taille = npos) const;

// Affectation :
basic_string &operator=(const basic_string &source);
basic_string &operator=(const charT *source);
basic_string &operator=(charT caractere);
basic_string &assign(const basic_string &source);
basic_string &assign(const basic_string &source,
    size_type position, size_type nombre);
basic_string &assign(const charT *chaine, size_type nombre);
basic_string &assign(const charT *chaine);
basic_string &assign(size_type nombre, charT caractere);
template <class InputIterator>
basic_string &assign(InputIterator debut, InputIterator fin);

// Concaténation et ajout :
basic_string &operator+=(const basic_string &source);
basic_string &operator+=(const charT *chaine);
basic_string &operator+=(charT caractere);
basic_string &append(const basic_string &source);
basic_string &append(const basic_string &source,
    size_type position, size_type nombre);
basic_string &append(const charT *chaine, size_type nombre);
basic_string &append(const charT *chaine);
basic_string &append(size_type nombre, charT caractere);
template <class InputIterator>
basic_string &append(InputIterator debut, InputIterator fin);

// Insertion et extraction :
basic_string &insert(size_type position, const basic_string &source);
basic_string &insert(size_type position, const basic_string &source,
    size_type debut, size_type nombre);
basic_string &insert(size_type position, const charT *chaine,
    size_type nombre);
basic_string &insert(size_type position, const charT *chaine);
basic_string &insert(size_type position, size_type nombre,
    charT caractere);
iterator insert(iterator position, charT caractere = charT());
void insert(iterator position, size_type nombre, charT caractere);
template <class InputIterator>
void insert(iterator position, InputIterator debut, InputIterator fin);

// Suppression :
basic_string &erase(size_type debut = 0, size_type longueur = npos);
iterator erase(iterator position);
iterator erase(iterator debut, iterator fin);
void clear();

// Remplacement et échange :
basic_string &replace(size_type position, size_type longueur,
    const basic_string &remplacement);
basic_string &replace(size_type position, size_type longueur,
    const basic_string &remplacement, size_type debut,
    size_type taille);
basic_string &replace(size_type position, size_type longueur,
    const charT *remplacement, size_type taille);
```

```
basic_string &replace(size_type position, size_type longueur,
    const charT *remplacement);
basic_string &replace(size_type position, size_type longueur,
    size_type nombre, charT caractere);
basic_string &replace(iterator debut, iterator fin,
    const basic_string &remplacement);
basic_string &replace(iterator debut, iterator fin,
    const charT *remplacement, size_type taille);
basic_string &replace(iterator debut, iterator fin,
    const charT *remplacement);
basic_string &replace(iterator debut, iterator fin,
    size_type nombre, charT caractere);
template <class InputIterator>
basic_string &replace(iterator debut, iterator fin,
    InputIterator debut_remplacement, InputIterator fin_remplacement);
void swap(basic_string<charT, traits, Allocator> &chaine);

// Comparaison :
int compare(const basic_string &chaine) const;
int compare(size_type debut1, size_type longueur1,
    const basic_string &chaine,
    size_type debut2, size_type longueur2) const;
int compare(const charT *chaine) const;
int compare(size_type debut, size_type longueur, const charT *chaine,
    size_type taille = npos) const;

// Recherche :
size_type find(const basic_string &motif,
    size_type position = 0) const;
size_type find(const charT *motif, size_type position,
    size_type taille) const;
size_type find(const charT *motif, size_type position = 0) const;
size_type find(charT caractere, size_type position = 0) const;
size_type rfind(const basic_string &motif,
    size_type position = npos) const;
size_type rfind(const charT *motif, size_type position,
    size_type taille) const;
size_type rfind(const charT *motif, size_type position = npos) const;
size_type rfind(charT caractere, size_type position = npos) const;
size_type find_first_of(const basic_string &motif,
    size_type position = 0) const;
size_type find_first_of(const charT *motif, size_type position,
    size_type taille) const;
size_type find_first_of(const charT *motif,
    size_type position = 0) const;
size_type find_first_of(charT caractere, size_type position = 0) const;
size_type find_last_of(const basic_string &motif,
    size_type position = npos) const;
size_type find_last_of(const charT *motif, size_type position,
    size_type taille) const;
size_type find_last_of(const charT *motif,
    size_type position = npos) const;
size_type find_last_of(charT caractere,
    size_type position = npos) const;
size_type find_first_not_of(const basic_string &motif,
    size_type position = 0) const;
size_type find_first_not_of(const charT *motif, size_type position,
```

```
        size_type taille) const;
    size_type find_first_not_of(const charT *motif,
        size_type position = 0) const;
    size_type find_first_not_of(charT caractere,
        size_type position = 0) const;
    size_type find_last_not_of(const basic_string &motif,
        size_type position = npos) const;
    size_type find_last_not_of(const charT *motif, size_type position,
        size_type taille) const;
    size_type find_last_not_of(const charT *motif,
        size_type position = npos) const;
    size_type find_last_not_of(charT caractere,
        size_type position = npos) const;
};

typedef basic_string<char>    string;
typedef basic_string<wchar_t> wstring;
```

Les opérateurs de concaténation, de comparaison et de sérialisation dans les flux d'entrée / sortie sont également définis dans cet en-tête et n'ont pas été reportés ici par souci de clarté. Comme vous pouvez le voir, la classe basic_string dispose d'un grand nombre de méthodes. Nous allons à présent les détailler dans les paragraphes suivants.

La bibliothèque standard définit deux types chaînes de caractères pour les types standards de caractères du langage : le type string pour les char, et le type wstring pour les wchar_t. En pratique, ce seront donc ces types qui seront utilisés dans les programmes. Les exemples de la suite de ce document utiliseront donc le type string, mais vous êtes libre d'utiliser des instances de la classe basic_string pour d'autres types de caractères.

14.1.1. Construction et initialisation d'une chaîne

La manière la plus simple de construire une basic_string est simplement de la déclarer, sans paramètres. Cela a pour conséquence d'appeler le constructeur par défaut, et d'initialiser la chaîne à la chaîne vide.

En revanche, si vous désirez initialiser cette chaîne, plusieurs possibilités s'offrent à vous. Outre le constructeur de copie, qui permet de copier une autre basic_string, il existe plusieurs surcharges du constructeur permettant d'initialiser la chaîne de différentes manières. Le constructeur le plus utilisé sera sans aucun doute le constructeur qui prend en paramètre une chaîne de caractères C classique :

```
string chaine("Valeur initiale");
```

Il existe cependant une variante de ce constructeur, qui prend en paramètre le nombre de caractères de la chaîne source à utiliser pour l'initialisation de la basic_string. Ce constructeur devra être utilisé dans le cas des tableaux de caractères, qui contiennent des chaînes de caractères qui ne sont pas nécessairement terminées par un caractère nul :

```
string chaine("Valeur initiale", 6);
```

La ligne précédente initialise la chaîne `chaine` avec la chaîne de caractères `"Valeur"`, car seuls les six premiers caractères de la chaîne d'initialisation sont utilisés.

Il est également possible d'initialiser une basic_string avec une partie de chaîne de caractères seulement. Pour cela, il faut utiliser le constructeur `template`, qui prend en paramètre un itérateur référençant le premier caractère à utiliser lors de l'initialisation de la basic_string et un itérateur référençant le

caractère suivant le dernier caractère à utiliser. Bien entendu, ces deux itérateurs sont de simples pointeurs de caractères si les caractères devant servir à l'initialisation sont dans une chaîne de caractères C ou dans un tableau de caractères. Cependant, ce peut être des itérateurs d'un conteneur quelconque, pourvu que celui-ci contienne bien une séquence de caractères et que le deuxième itérateur se trouve bien après le premier dans cette séquence. Notez que le deuxième itérateur ne référence pas le dernier caractère de la séquence d'initialisation, mais bien le caractère suivant. Il peut donc valoir la valeur de fin de l'itérateur du conteneur source. Ainsi, le code suivant :

```
char *source = "Valeur initiale";
string s(source, source+6);
```

a strictement le même effet que celui de l'exemple précédent.

Enfin, il existe un constructeur dont le but est d'initialiser une basic_string avec une suite de caractères identiques d'une certaine longueur. Ce constructeur n'est réellement pas difficile à utiliser, puisqu'il suffit de lui fournir en paramètre le nombre de caractères que la basic_string devra contenir et la valeur du caractère qui devra être répété dans cette chaîne.

Vous remarquerez que tous ces constructeurs prennent en dernier paramètre une instance d'allocateur mémoire à utiliser pour les opérations d'allocation et de libération de la mémoire que la chaîne est susceptible d'avoir à faire. Vous pouvez spécifier une instance quelconque, ou utiliser la valeur par défaut fournie par les déclarations des constructeurs. Cette valeur par défaut est tout simplement une instance temporaire de l'allocateur spécifié en paramètre `template` de la classe basic_string. Par défaut, cet allocateur est l'allocateur standard, pour lequel toutes les instances permettent d'accéder à la même mémoire. Il n'est donc pas nécessaire de spécifier l'instance à utiliser, puisqu'elles sont toutes fonctionnellement identiques. En pratique donc, il est très rare d'avoir à spécifier un allocateur mémoire dans ces constructeurs.

14.1.2. Accès aux propriétés d'une chaîne

La classe basic_string fournit un certain nombre d'accesseurs permettant d'obtenir des informations sur son état et sur la chaîne de caractères qu'elle contient. L'une des informations les plus intéressantes est sans nul doute la longueur de cette chaîne. Elle peut être obtenue à l'aide de deux accesseurs, qui sont strictement équivalents : `size` et `length`. Vous pouvez utiliser l'un ou l'autre, selon votre convenance. Par ailleurs, si vous désirez simplement savoir si la basic_string est vide, vous pouvez appeler la méthode `empty`.

> **Note :** Attention, contrairement à ce que son nom pourrait laisser penser, la méthode `empty` ne vide pas la basic_string !

La taille maximale qu'une basic_string peut contenir est souvent directement liée à la quantité de mémoire disponible, puisque la chaîne de caractères qu'elle contient est allouée dynamiquement. Il n'y a donc souvent pas beaucoup d'intérêt à obtenir cette taille, mais vous pouvez malgré tout le faire, grâce à la méthode `max_size`.

La quantité de mémoire réellement allouée par une basic_string peut être supérieure à la longueur de la chaîne de caractères contenue. En effet, la classe basic_string peut conserver une marge de manœuvre, pour le cas où la chaîne devrait être agrandie à la suite d'une opération particulière. Cela permet de réduire les réallocations de mémoire, qui peuvent être très coûteuses lorsque la mémoire se fragmente (la chaîne doit être recopiée vers un nouvel emplacement si un autre bloc mémoire se trouve juste après le bloc mémoire à réallouer). Cette quantité de mémoire peut être obtenue à l'aide

de la méthode `capacity`. Nous verrons comment réserver de la place mémoire en prévision d'un redimensionnement ultérieur dans la section suivante.

Dans le cas où vous utiliseriez un allocateur différent de l'allocateur par défaut, vous pouvez obtenir une copie de cet allocateur grâce à la méthode `get_allocator`. Il est relativement rare d'avoir à utiliser cette méthode.

14.1.3. Modification de la taille des chaînes

Une fois qu'une basic_string a été construite, il est possible de la modifier a posteriori pour la réduire, l'agrandir ou augmenter sa capacité. Ces opérations peuvent être réalisées à l'aide de méthodes fournies à cet effet.

La méthode `resize` permet de modifier la taille de la chaîne de caractères stockée dans la basic_string. Dans sa version la plus simple, elle prend en paramètre la nouvelle taille que la chaîne doit avoir. Si cette taille est inférieure à la taille courante, la chaîne est tronquée. En revanche, si cette taille est supérieure à la taille actuelle, la chaîne est étendue et les nouveaux caractères sont initialisés avec la valeur des caractères définie par le constructeur par défaut de leur classe. Pour les types prédéfinis char et wchar_t, cette valeur est toujours le caractère nul. Les données stockées dans la basic_string représentent donc toujours la même chaîne de caractères C, puisque ces chaînes utilisent le caractère nul comme marqueur de fin de chaîne. Ainsi, la longueur renvoyée par la méthode `size` peut être différente de la longueur de la chaîne C contenue par la basic_string.

Exemple 14-1. Redimensionnement d'une chaîne

```
#include <iostream>
#include <string>
#include <string.h>

using namespace std;

int main(void)
{
    string s("123");
    s.resize(10);
    cout << s << endl;
    // La nouvelle taille vaut bien 10 :
    cout << "Nouvelle taille : " << s.length() << endl;
    // mais la longueur de la chaîne C reste 3 :
    cout << "Longueur C : " << strlen(s.c_str()) << endl;
    return 0;
}
```

> **Note :** La méthode `c_str` utilisée dans cet exemple sera décrite en détail dans la section suivante. Elle permet d'obtenir l'adresse de la chaîne C stockée en interne dans la basic_string.
>
> La fonction `strlen` quant à elle est une des fonctions de manipulation des chaînes de caractères de la bibliothèque C. Elle est définie dans le fichier d'en-tête `string.h` (que l'on ne confondra pas avec l'en-tête `string` de la bibliothèque standard C++), et renvoie la longueur de la chaîne de caractères qui lui est fournie en paramètre.

Si la valeur par défaut utilisée pour les caractères complémentaires dans la méthode `resize` n'est pas celle qui est désirée, il faut en utiliser une autre version. Cette deuxième version prend, en plus

de la nouvelle taille de la chaîne de caractères, le caractère de remplissage à utiliser pour le cas où la nouvelle taille serait supérieure à la taille initiale de la chaîne :

```
string s("123");
s.resize(10, 'a');
```

Dans cet exemple, `s` contient finalement la chaîne de caractères `"123aaaaaaa"`.

Nous avons vu précédemment que les basic_string étaient susceptibles d'allouer plus de mémoire que nécessaire pour stocker leurs données afin de limiter le nombre de réallocation mémoire. Ce mécanisme est complètement pris en charge par la bibliothèque, et le programmeur n'a en général pas à s'en soucier. Cependant, il peut exister des situations où l'on sait à l'avance la taille minimale qu'une chaîne doit avoir pour permettre de travailler dessus sans craindre de réallocations mémoire successives. Dans ce cas, on a tout intérêt à fixer la capacité de la chaîne directement à cette valeur, afin d'optimiser les traitements. Cela est réalisable à l'aide de la méthode `reserve`. Cette méthode prend en paramètre la capacité minimale que la basic_string doit avoir. La nouvelle capacité n'est pas forcément égale à ce paramètre après cet appel, car la basic_string peut allouer plus de mémoire que demandé.

Exemple 14-2. Réservation de mémoire dans une chaîne

```
#include <iostream>
#include <string>

using namespace std;

int main(void)
{
    string s;
    cout << s.capacity() << endl;
    s.reserve(15);
    s = "123";
    cout << s.capacity() << endl;
    return 0;
}
```

> **Note :** Les méthodes `resize` et `reserve` peuvent effectuer une réallocation de la zone mémoire contenant la chaîne de caractères. Par conséquent, toutes les références sur les caractères de la chaîne et tous les itérateurs sur cette chaîne deviennent invalide à la suite de leur exécution.

14.1.4. Accès aux données de la chaîne de caractères

Les caractères des basic_string peuvent être accédés de nombreuses manières. Premièrement, la classe basic_string surcharge l'opérateur d'accès aux éléments d'un tableau, et l'on pourra les utiliser pour obtenir une référence à un des caractères de la chaîne à partir de son indice. Cet opérateur n'est défini que pour les indices valides dans la chaîne de caractères, à savoir les indices variant de 0 à la valeur retournée par la méthode `size` de la chaîne moins un :

```
#include <iostream>
#include <string>
```

```
using namespace std;

int main(void)
{
    string s("azc");
    // Remplace le deuxième caractère de la chaîne par un 'b' :
    s[1] = 'b';
    cout << s << endl;
    return 0;
}
```

Lorsqu'il est appliqué à une basic_string constante, l'opérateur tableau peut renvoyer la valeur du caractère dont l'indice est exactement la taille de la chaîne. Il s'agit évidemment du caractère nul servant de marqueur de fin de chaîne. En revanche, la référence renvoyée par cet opérateur pour toutes les autres valeurs, ainsi que par l'opérateur tableau appliqué aux chaînes non constante pour le caractère de fin de chaîne ne sont pas valides. Le comportement des programmes qui effectuent de tels accès est imprévisible.

Il existe une autre possibilité pour accéder aux caractères d'une basic_string. Il s'agit de la méthode `at`. Contrairement à l'opérateur tableau, cette méthode permet d'effectuer un contrôle sur la validité de l'indice utilisé. Elle renvoie, comme l'opérateur de tableau de la classe basic_string, la référence du caractère dont l'indice est spécifié en paramètre. Cependant, elle effectue au préalable un contrôle sur la validité de cet indice, qui doit toujours être strictement inférieur à la taille de la chaîne. Dans le cas contraire, la méthode `at` lance une exception out_of_range :

```
#include <iostream>
#include <string>
#include <stdexcept>

using namespace std;

int main(void)
{
    string s("01234");
    try
    {
        s.at(5);
    }
    catch (const out_of_range &)
    {
        cout << "Débordement !" << endl;
    }
    return 0;
}
```

La classe basic_string ne contient pas d'opérateur de transtypage vers les types des chaînes de caractères C classique, à savoir le type pointeur sur caractère et pointeur sur caractère constant. C'est un choix de conception, qui permet d'éviter les conversions implicites des basic_string en chaîne C classique, qui pourraient être extrêmement dangereuses. En effet, ces conversions conduiraient à obtenir implicitement des pointeurs qui ne seraient plus valides dès qu'une opération serait effectuée sur la basic_string source. Cependant, la classe basic_string fournit deux méthodes permettant d'obtenir de tels pointeurs de manière explicite. Le programmeur prend donc ses responsabilités lorsqu'il utilise ces méthodes, et est supposé savoir dans quel contexte ces pointeurs sont valides.

Ces deux méthodes sont respectivement la méthode `data` et la méthode `c_str`. La première méthode renvoie un pointeur sur les données de la basic_string. Ces données ne sont rien d'autre qu'un tableau de caractères, dont la dimension est exactement la valeur retournée par la méthode `size` ou par la méthode `length`. Ce tableau peut contenir des caractères de terminaison de chaîne, si par exemple une telle valeur a été écrite explicitement ou a été introduite suite à un redimensionnement de la chaîne. La méthode `c_str` en revanche retourne un pointeur sur la chaîne de caractères C contenue dans la basic_string. Contrairement aux données renvoyées par la méthode `data`, cette chaîne est nécessairement terminée par un caractère de fin de chaîne. Cette méthode sera donc utilisée partout où l'on veut obtenir une chaîne de caractères C classique, mais elle ne devra pas être utilisée pour accéder aux données situées après ce caractère de fin de chaîne.

Exemple 14-3. Accès direct aux données d'une chaîne

```
#include <iostream>
#include <string>

using namespace std;

int main(void)
{
    string s("123456");
    // La chaîne C est coupée au troisième caractère :
    s[3] = 0;
    cout << s.c_str() << endl;
    // Mais on peut quand même obtenir les caractères suivants :
    cout << s.data()[4] << s.data()[5] << endl;
    return 0;
}
```

Notez que ces méthodes renvoient des pointeurs sur des données constantes. Cela est normal, car il est absolument interdit de modifier les données internes à la basic_string qui a fourni ces pointeurs. Vous ne devez donc en aucun cas essayer d'écrire dans ces tableaux de caractères, même en faisant un transtypage au préalable.

Enfin, la classe basic_string définit des itérateurs à accès aléatoires permettant d'accéder à ses données comme s'il s'agissait d'une chaîne de caractères standard. Les méthodes `begin` et `end` permettent d'obtenir respectivement un itérateur sur le premier caractère de la chaîne et la valeur de fin de ce même itérateur. De même, les méthodes `rbegin` et `rend` permettent de parcourir les données de la basic_string en sens inverse. Prenez garde au fait que ces itérateurs ne sont valides que tant qu'aucune méthode modifiant la chaîne n'est appelée.

14.1.5. Opérations sur les chaînes

La classe basic_string fournit tout un ensemble de méthodes permettant d'effectuer les opérations les plus courantes sur les chaînes de caractères. C'est cet ensemble de méthodes qui font tout l'intérêt de cette classe par rapport aux chaînes de caractères classiques du langage C, car elles prennent en charge automatiquement les allocations mémoire et les copies de chaînes que l'implémentation de ces fonctionnalités peut imposer. Ces opérations comprennent l'affectation et la concaténation de deux chaînes de caractères, l'extraction d'une sous-chaîne, ainsi que la suppression et le remplacement de caractères dans une chaîne.

14.1.5.1. Affectation et concaténation de chaînes de caractères

Plusieurs surcharges de l'opérateur d'affectation sont définies dans la classe basic_string. Ces surcharges permettent d'affecter une nouvelle valeur à une chaîne, en remplaçant éventuellement l'ancienne valeur. Dans le cas d'un remplacement, la mémoire consommée par l'ancienne valeur est automatiquement réutilisée ou libérée si une réallocation est nécessaire. Ces opérateurs d'affectation peuvent prendre en paramètre une autre basic_string, une chaîne de caractères C classique ou même un simple caractère. Leur utilisation est donc directe, il suffit simplement d'écrire une affectation normale.

Il est impossible, avec l'opérateur d'affectation, de fournir des paramètres supplémentaires comme ceux dont les constructeurs de la classe basic_string disposent par exemple. C'est pour cette raison qu'une autre méthode, la méthode `assign`, a été définie pour permettre de faire des affectations plus complexes.

Les premières versions de ces méthodes permettent bien entendu d'effectuer l'affectation d'une autre basic_string ou d'une chaîne de caractères C classique. Cependant, il est également possible de spécifier la longueur de la portion de chaîne à copier en deuxième paramètre pour les chaînes C, et la position ainsi que le nombre de caractères à copier dans le cas d'une basic_string. Il est également possible de réinitialiser une basic_string avec un caractère spécifique, en donnant et le nombre de caractères à dupliquer dans la chaîne en premier paramètre et la valeur de ce caractère en deuxième paramètre. Enfin, il existe une version `template` de cette méthode permettant d'affecter à la basic_string la séquence de caractères compris entre deux itérateurs d'un conteneur.

Exemple 14-4. Affectation de chaîne de caractères

```
#include <iostream>
#include <string>

using namespace std;

int main(void)
{
    string s1, s2;
    char *p="1234567890";
    // Affecte "345" à s1 :
    s1.assign(p+2, p+6);
    cout << s1 << endl;
    // Affecte les deux premiers caractères de s1 à s2 :
    s2.assign(s1, 0, 2);
    cout << s2 << endl;
    // Réinitialise s1 avec des 'A' :
    s1.assign(5, 'A');
    cout << s1 << endl;
    return 0;
}
```

De manière similaire, l'opérateur d'affectation avec addition '+=' a été surchargé afin de permettre les concaténations de chaînes de caractères de manière intuitive. Cet opérateur permet d'ajouter aussi bien une autre basic_string qu'une chaîne de caractères C classique ou un unique caractère à la fin d'une basic_string. Comme cet opérateur est trop restrictif lorsqu'il s'agit de concaténer une partie seulement d'une autre chaîne à une basic_string, un jeu de méthodes `append` a été défini. Ces méthodes permettent d'ajouter à une basic_string une autre basic_string ou une chaîne de caractères C bien entendu, mais aussi d'ajouter une partie seulement de ces chaînes ou un nombre déterminé de caractères. Toutes ces méthodes prennent les mêmes paramètres que les méthodes `assign` correspondantes et leur emploi ne devrait pas poser de problème particulier.

Exemple 14-5. Concaténation de chaînes de carctères

```
#include <iostream>
#include <string>

using namespace std;

int main(void)
{
    string s1 = "abcef";
    string s2 = "ghijkl";
    // Utilisation de l'opérateur de concaténation :
    s1+=s2;
    cout << s1 << endl;
    // Utilisation de la méthode append :
    s1.append("mnopq123456", 5);
    cout << s1 << endl;
    return 0;
}
```

14.1.5.2. Extraction de données d'une chaîne de caractères

Nous avons vu que les méthodes `data` et `c_str` permettaient d'obtenir des pointeurs sur les données des basic_string. Cependant, il est interdit de modifier les données de la basic_string au travers de ces pointeurs. Or, il peut être utile, dans certaines situations, d'avoir à travailler sur ces données, il faut donc pouvoir en faire une copie temporaire. C'est ce que permet la méthode `copy`. Cette fonction prend en paramètre un pointeur sur une zone de mémoire devant accueillir la copie des données de la basic_string, le nombre de caractères à copier, ainsi que le numéro du caractère de la basic_string à partir duquel la copie doit commencer. Ce dernier paramètre est facultatif, car il prend par défaut la valeur 0 (la copie se fait donc à partir du premier caractère de la basic_string.

Exemple 14-6. Copie de travail des données d'une basic_string

```
#include <iostream>
#include <string>

using namespace std;

int main(void)
{
    string s="1234567890";
    // Copie la chaîne de caractères dans une zone de travail :
    char buffer[16];
    s.copy(buffer, s.size(), 0);
    buffer[s.size()] = 0;
    cout << buffer << endl;
    return 0;
}
```

La basic_string doit contenir suffisamment de caractères pour que la copie puisse se faire. Si ce n'est pas le cas, elle lancera une exception out_of_range. En revanche, la méthode `copy` ne peut faire aucune vérification quant à la place disponible dans la zone mémoire qui lui est passée en paramètre. Il est donc de la responsabilité du programmeur de s'assurer que cette zone est suffisamment grande pour accueillir tous les caractères qu'il demande.

La méthode `copy` permet d'obtenir la copie d'une sous-chaîne de la chaîne contenue dans la basic_string. Toutefois, si l'on veut stocker cette sous-chaîne dans une autre basic_string, il ne faut pas utiliser cette méthode. La méthode `substr` permet en effet d'effectuer ce travail directement. Cette méthode prend en premier paramètre le numéro du premier caractère à partir de la sous-chaîne à copier, ainsi que sa longueur. Comme pour la méthode `copy`, il faut que la basic_string source contienne suffisamment de caractères faute de quoi une exception out_of_range sera lancée.

Exemple 14-7. Extraction de sous-chaîne

```
#include <iostream>
#include <string>

using namespace std;

int main(void)
{
    string s1 = "1234567890";
    string s2 = s1.substr(2, 5);
    cout << s2 << endl;
    return 0;
}
```

14.1.5.3. Insertion et suppression de caractères dans une chaîne

La classe basic_string dispose de tout un jeu de méthodes d'insertion de caractères ou de chaînes de caractères au sein d'une basic_string existante. Toutes ces méthodes sont des surcharges de la méthode `insert`. Ces surcharges prennent toutes un paramètre en première position qui indique l'endroit où l'insertion doit être faite. Ce paramètre peut être soit un numéro de caractère, indiqué par une valeur de type size_type, soit un itérateur de la basic_string dans laquelle l'insertion doit être faite. Les autres paramètres permettent de spécifier ce qui doit être inséré dans cette chaîne.

Les versions les plus simples de la méthode `insert` prennent en deuxième paramètre une autre basic_string ou une chaîne de caractères C classique. Leur contenu sera inséré à l'emplacement indiqué. Lorsque le deuxième paramètre est une basic_string, il est possible d'indiquer le numéro du premier caractère ainsi que le nombre de caractères total à insérer. De même, lors de l'insertion d'une chaîne C classique, il est possible d'indiquer le nombre de caractères de cette chaîne qui doivent être insérés.

Il existe aussi des méthodes `insert` permettant d'insérer un ou plusieurs caractères à un emplacement donné dans la chaîne de caractères. Ce caractère doit alors spécifié en deuxième paramètre, sauf si l'on veut insérer plusieurs caractères identiques dans la chaîne, auquel cas on doit indiquer le nombre de caractères à insérer et la valeur de ce caractère.

Enfin, il existe une version de la méthode `insert` qui prend en paramètre, en plus de l'itérateur spécifiant la position à partir de laquelle l'insertion doit être faite dans la basic_string, deux autres itérateurs d'un quelconque conteneur contenant des caractères. Utilisé avec des pointeurs de caractères, cet itérateur peut être utilisé pour insérer un morceau quelconque de chaîne de caractères C dans une basic_string.

Toutes ces méthodes renvoient généralement la basic_string sur laquelle ils travaillent, sauf les méthodes qui prennent en paramètre un itérateur. Ces méthodes supposent en effet que la manipulation de la chaîne de caractères se fait par l'intermédiaire de cet itérateur, et non par l'intermédiaire d'une référence sur la basic_string. Cependant, la méthode `insert` permettant d'insérer un caractère unique à un emplacement spécifié par un itérateur renvoie la valeur de l'itérateur référençant le caractère qui vient d'être inséré afin de permettre de récupérer ce nouvel itérateur pour les opérations ultérieures.

Exemple 14-8. Insertion de caractères dans une chaîne

```
#include <iostream>
#include <string>

using namespace std;

int main(void)
{
    string s = "abef";
    // Insère un 'c' et un 'd' :
    s.insert(2, "cdze", 2);
    // Idem pour 'g' et 'h', mais avec une basic_string :
    string gh = "gh";
    s.insert(6, gh);
    cout << s << endl;
    return 0;
}
```

Il existe également trois surcharges de la méthode `erase`, dont le but est de supprimer des caractères dans une chaîne en décalant les caractères suivant les caractères supprimés pour remplir les positions ainsi libérées. La première méthode `erase` prend en premier paramètre la position du premier caractère et le nombre des caractères à supprimer. La deuxième méthode fonctionne de manière similaire, mais prend en paramètre l'itérateur de début et l'itérateur de fin de la sous-chaîne de caractères qui doit être supprimée. Enfin, la troisième version de `erase` permet de supprimer un unique caractère, dont la position est spécifiée encore une fois par un itérateur. Ces deux dernières méthodes renvoient l'itérateur référençant le caractère suivant le dernier caractère qui a été supprimé de la chaîne. S'il n'y avait pas de caractères après le dernier caractère effacé, l'itérateur de fin de chaîne est renvoyé.

Enfin, il existe une méthode dédiée pour l'effacement complet de la chaîne de caractères contenue dans une basic_string. Cette méthode est la méthode `clear`.

Exemple 14-9. Suppression de caractères dans une chaîne

```
#include <iostream>
#include <string>

using namespace std;

int main(void)
{
    string s = "abcdeerrfgh";
    // Supprime la faute de frappe :
    s.erase(5,3);
    cout << s << endl;
    // Efface la chaîne de caractères complète :
    s.clear();
    if (s.empty()) cout << "Vide !" << endl;
    return 0;
}
```

14.1.5.4. Remplacements de caractères d'une chaîne

Comme pour l'insertion de chaînes de caractères, il existe tout un jeu de fonctions permettant d'effectuer un remplacement d'une partie de la chaîne de caractères stockée dans les basic_string par une autre chaîne de caractères. Ces méthodes sont nommées `replace` et sont tout à fait similaires dans le principe aux méthodes `insert`. Cependant, contrairement à celles-ci, les méthodes `replace` prennent un paramètre supplémentaire pour spécifier la longueur ou le caractère de fin de la sous-chaîne à remplacer. Ce paramètre doit être fourni juste après le premier paramètre, qui indique toujours le caractère de début de la sous-chaîne à remplacer. Il peut être de type size_type pour les versions de `replace` qui travaillent avec des indices, ou être un itérateur, pour les versions de `replace` qui travaillent avec des itérateurs. Les autres paramètres des fonctions `replace` permettent de décrire la chaîne de remplacement, et fonctionnent exactement comme les paramètres correspondants des fonctions `insert`.

Exemple 14-10. Remplacement d'une sous-chaîne dans une chaîne

```
#include <iostream>
#include <string>

using namespace std;

int main(void)
{
    string s = "abcerfg";
    // Remplace le 'e' et le 'r' par un 'd' et un 'e' :
    s.replace(3, 2, "de");
    cout << s << endl;
    return 0;
}
```

Dans le même ordre d'idée que le remplacement, on trouvera la méthode `swap` de la classe basic_string, qui permet d'intervertir le contenu de deux chaînes de caractères. Cette méthode prend en paramètre une référence sur la deuxième chaîne de caractères, avec laquelle l'interversion doit être faite. La méthode `swap` pourra devra être utilisée de préférence pour réaliser les échanges de chaînes de caractères, car elle est optimisée et effectue en fait l'échange par référence. Elle permet donc d'éviter de faire une copie temporaire de la chaîne destination et d'écraser la chaîne source avec cette copie.

Exemple 14-11. Échange du contenu de deux chaînes de caractères

```
#include <iostream>
#include <string>

using namespace std;

int main(void)
{
    string s1 = "abcd";
    string s2 = "1234";
    cout << "s1 = " << s1 << endl;
    cout << "s2 = " << s2 << endl;
    // Intervertit les deux chaînes :
    s1.swap(s2);
    cout << "s1 = " << s1 << endl;
    cout << "s2 = " << s2 << endl;
```

```
    return 0;
}
```

14.1.6. Comparaison de chaînes de caractères

La comparaison des basic_string se base sur la méthode `compare`, dont plusieurs surcharges existent afin de permettre des comparaisons diverses et variées. Les deux versions les plus simples de la méthode `compare` prennent en paramètre soit une autre basic_string, soit une chaîne de caractères C classique. Elles effectuent donc la comparaison de la basic_string sur laquelle elles s'appliquent avec ces chaînes. Elles utilisent pour cela la méthode `eq` de la classe des traits des caractères utilisés par la chaîne. Si les deux chaînes ne diffèrent que par leur taille, la chaîne la plus courte sera déclarée inférieure à la chaîne la plus longue.

Les deux autres méthodes `compare` permettent d'effectuer la comparaison de sous-chaînes de caractères entre elles. Elles prennent toutes les deux l'indice du caractère de début et l'indice du caractère de fin de la sous-chaîne de la basic_string sur laquelle elles sont appliquées, un troisième paramètre indiquant une autre chaîne de caractères, et des indices spécifiant la deuxième sous-chaîne dans cette chaîne. Si le troisième argument est une basic_string, il faut spécifier également l'indice de début et l'indice de fin de la sous-chaîne. En revanche, s'il s'agit d'une chaîne C classique, la deuxième sous-chaîne commence toujours au premier caractère de cette chaîne, et il ne faut spécifier que la longueur de cette sous-chaîne.

La valeur renvoyée par les méthodes `compare` est de type entier. Cet entier est nul si les deux chaînes sont strictement égales (et de même taille), négatif si la basic_string sur laquelle la méthode `compare` est appliquée est plus petite que la chaîne passée en argument, soit en taille, soit au sens de l'ordre lexicographique, et positif dans le cas contraire.

Exemple 14-12. Comparaisons de chaînes de caractères

```
#include <iostream>
#include <string>

using namespace std;

int main(void)
{
    const char *c1 = "bcderefb";
    const char *c2 = "bcdetab";   // c2 > c1
    const char *c3 = "bcderefas"; // c3 < c1
    const char *c4 = "bcde";      // c4 < c1
    string s1 = c1;
    if (s1 < c2) cout << "c1 < c2" << endl;
    else cout << "c1 >= c2" << endl;
    if (s1.compare(c3)>0) cout << "c1 > c3" << endl;
    else cout << "c1 <= c3" << endl;
    if (s1.compare(0, string::npos, c1, 4)>0)
        cout << "c1 > c4" << endl;
    else cout << "c1 <= c4" << endl;
    return 0;
}
```

Bien entendu, les opérateurs de comparaison classiques sont également définis afin de permettre des comparaisons simples entre chaîne de caractères. Grâce à ces opérateurs, il est possible de manipuler

les basic_string exactement comme les autres types ordonnés du langage. Plusieurs surcharge de ces opérateurs ont été définies et travaillent avec les différents types de données avec lesquels il est possible pour une basic_string de se comparer. L'emploi de ces opérateurs est naturel et ne pose pas de problèmes particuliers.

> **Note :** Toutes ces comparaisons se basent sur l'ordre lexicographique du langage C. Autrement dit, les comparaisons entre chaînes de caractères ne tiennent pas compte de la locale et des conventions nationales. Elles sont donc très efficaces, mais ne pourront pas être utilisées pour comparer des chaînes de caractères humainement lisibles. Vous trouverez de plus amples renseignements sur la manière de prendre en compte les locales dans les comparaisons de chaînes de caractères dans le Chapitre 16.

14.1.7. Recherche dans les chaînes

Les opérations de recherche dans les chaînes de caractères constituent une des fonctionnalités des chaînes les plus courantes. Elles constituent la plupart des opérations d'analyse des chaînes, et sont souvent le pendant de la construction et la concaténation de chaînes. La classe basic_string fournit donc tout un ensemble de méthodes permettant d'effectuer des recherches de caractères ou de sous-chaînes dans une basic_string.

Les fonctions de recherche sont toutes surchargées afin de permettre de spécifier la position à partir de laquelle la recherche doit commencer d'une part, et le motif de caractère à rechercher. Le premier paramètre indique toujours quel est ce motif, que ce soit une autre basic_string, une chaîne de caractères C classique ou un simple caractère. Le deuxième paramètre est le numéro du caractère de la basic_string sur laquelle la méthode de recherche s'applique et à partir duquelle elle commence. Ce deuxième paramètre peut être utilisé pour effectuer plusieurs recherches successives, en repartant de la dernière position trouvée à chaque fois. Lors d'une première recherche ou lors d'une recherche unique, il n'est pas nécessaire de donner la valeur de ce paramètre, car les méthodes de recherche utilisent la valeur par défaut qui convient (soit le début de la chaîne, soit la fin, selon le sens de recherche utilisé par la méthode). Les paramètres suivants permettent de donner des informations complémentaires sur le motif à utiliser pour la recherche. Il n'est utilisé que lorsque le motif est une chaîne de caractères C classique. Dans ce cas, il est en effet possible de spécifier la longueur du motif dans cette chaîne.

Les différentes fonctions de recherche disponibles sont présentées dans le tableau suivant :

Tableau 14-1. Fonctions de recherche dans les chaînes de caractères

Méthode	Description
`find`	Cette méthode permet de rechercher la sous-chaîne correspondant au motif passé en paramètre dans la basic_string sur laquelle elle est appliquée. Elle retourne l'indice de la première occurrence de ce motif dans la chaîne de caractères, ou la valeur `npos` si le motif n'y apparaît pas.
`rfind`	Cette méthode permet d'effectuer une recherche similaire à celle de la méthode `find`, mais en parcourant la chaîne de caractères en sens inverse. Notez bien que ce n'est pas le motif qui est inversé ici, mais le sens de parcours de la chaîne. Ainsi, la méthode `rfind` retourne l'indice de la dernière occurrence du motif dans la chaîne, ou la valeur `npos` si le motif n'a pas été trouvé.

Méthode	Description
`find_first_of`	Cette méthode permet de rechercher la première occurrence d'un des caractères présents dans le motif fourni en paramètre. Il ne s'agit donc plus d'une recherche de chaîne de caractères, mais de la recherche de tous les caractères d'un ensemble donné. La valeur retournée est l'indice du caractère trouvé, ou la valeur `npos` si aucun caractère du motif n'est détecté dans la chaîne.
`find_last_of`	Cette méthode est à la méthode `find_first_of` ce que `rfind` est à `find`. Elle effectue la recherche du dernier caractère de la basic_string qui se trouve dans la liste des caractères du motif fourni en paramètre. La valeur retournée est l'indice de ce caractère s'il existe, et `npos` sinon.
`find_first_not_of`	Cette méthode travaille en logique inverse par rapport à la méthode `find_first_of`. En effet, elle recherche le premier caractère de la basic_string qui *n'est pas* dans le motif fourni en paramètre. Elle renvoie l'indice de ce caractère, ou `npos` si celui-ci n'existe pas.
`find_last_not_of`	Cette méthode effectue le même travail que la méthode `find_first_not_of`, mais en parcourant la chaîne de caractères source en sens inverse. Elle détermine donc l'indice du premier caractère en partant de la fin qui ne se trouve pas dans le motif fourni en paramètre. Elle renvoie `npos` si aucun caractère ne correspond à ce critère.

Exemple 14-13. Recherches dans les chaînes de caractères

```
#include <iostream>
#include <string>

using namespace std;

int main(void)
{
    string s = "Bonjour  tout    le monde !";
    // Recherche le mot "monde" :
    string::size_type pos = s.find("monde");
    cout << pos << endl;
    // Recherche le mot "tout" en commençant par la fin :
    pos = s.rfind("tout");
    cout << pos << endl;
    // Décompose la chaîne en mots :
    string::size_type debut = s.find_first_not_of(" \t\n");
    while (debut != string::npos)
    {
        // Recherche la fin du mot suivant :
        pos = s.find_first_of(" \t\n", debut);
        // Affiche le mot :
        if (pos != string::npos)
            cout << s.substr(debut, pos - debut) << endl;
        else
            cout << s.substr(debut) << endl;
        debut = s.find_first_not_of(" \t\n", pos);
    }
    return 0;
}
```

Note : Toutes ces fonctions de recherche utilisent l'ordre lexicographique du langage C pour effectuer leur travail. Elles peuvent donc ne pas convenir pour effectuer des recherches dans des chaînes de caractères saisies par des humains, car elles ne prennent pas en compte la locale et les paramètres nationaux de l'utilisateur. La raison de ce choix est essentiellement la recherche de l'efficacité dans la bibliothèque standard. Nous verrons dans le Chapitre 16 la manière de procéder pour prendre en compte les paramètres nationaux au niveau des chaînes de caractères.

14.1.8. Fonctions d'entrée / sortie des chaînes de caractères

Pour terminer ce tour d'horizon des chaînes de caractères, signalons que la bibliothèque standard C++ fournit des opérateurs permettant d'effectuer des écritures et des lectures sur les flux d'entrée / sortie. Les opérateurs '<<' et '>>' sont donc surchargés pour les basic_string, et permettent de manipuler celles-ci comme des types normaux. L'opérateur '<<' permet d'envoyer le contenu de la basic_string sur le flux de sortie standard. L'opérateur '>>' lit les données du flux d'entrée standard, et les affecte à la basic_string qu'il reçoit en paramètre. Il s'arrête dès qu'il rencontre le caractère de fin de fichier, un espace, ou que la taille maximale des basic_string a été atteinte (cas improbable) :

```
#include <iostream>
#include <string>

using namespace std;

int main(void)
{
    string s1, s2;
    cin >> s1;
    cin >> s2;
    cout << "Premier mot : " << endl << s1 << endl;
    cout << "Deuxième mot : " << endl << s2 << endl;
    return 0;
}
```

Cependant, ces opérateurs peuvent ne pas s'avérer suffisants. En effet, l'une des principales difficultés dans les programmes qui manipulent des chaînes de caractères est de lire les données qui proviennent d'un flux d'entrée ligne par ligne. La notion de *ligne* n'est pas très claire, et dépend fortement de l'environnement d'exécution. La bibliothèque standard C++ suppose, quant à elle, que les lignes sont délimitées par un caractère spécial servant de marqueur spécial. Généralement, ce caractère est le caractère `'\n'`, mais il est également possible d'utiliser d'autres séparateurs.

Pour simplifier les opérations de lecture de textes constitués de lignes, la bibliothèque fournit la fonction `getline`. Cette fonction prend en premier paramètre le flux d'entrée sur lequel elle doit lire la ligne, et la basic_string dans laquelle elle doit stocker cette ligne en deuxième paramètre. Le troisième paramètre permet d'indiquer le caractère séparateur de ligne. Ce paramètre est facultatif, car il dispose d'une valeur par défaut qui correspond au caractère de fin de ligne classique `'\n'`.

Exemple 14-14. Lecture de lignes sur le flux d'entrée

```
#include <iostream>
#include <string>
```

```
using namespace std;

int main(void)
{
    string s1, s2;
    getline(cin, s1);
    getline(cin, s2);
    cout << "Première ligne : " << s1 << endl;
    cout << "Deuxième ligne : " << s2 << endl;
    return 0;
}
```

14.2. Les types utilitaires

La bibliothèque standard utilise en interne un certain nombre de types de données spécifiques, essentiellement dans un but de simplicité et de facilité d'écriture. Ces types seront en général rarement utilisés par les programmeurs, mais certaines fonctionnalités de la bibliothèque standard peuvent y avoir recours. Il faut donc connaître leur existence et savoir les manipuler correctement.

14.2.1. Les pointeurs auto

La plupart des variables détruisent leur contenu lorsqu'elles sont détruites elles-mêmes. Une exception notable à ce comportement est bien entendu celle des pointeurs, qui par définition ne contiennent pas eux-mêmes leurs données mais plutôt une référence sur celles-ci. Lorsque ces données sont allouées dynamiquement, il faut systématiquement penser à les détruire manuellement lorsqu'on n'en a plus besoin. Cela peut conduire à des *fuites de mémoire* (« Memory Leak » en anglais) très facilement. Si de telles fuites ne sont pas gênantes pour les processus dont la durée de vie est très courte, elles peuvent l'être considérablement plus pour les processus destinés à fonctionner longtemps, si ce n'est en permanence, sur une machine.

En fait, dans un certain nombre de cas, l'allocation dynamique de mémoire n'est utilisée que pour effectuer localement des opérations sur un nombre arbitraire de données qui ne peut être connu qu'à l'exécution. Cependant, il est relativement rare d'avoir à conserver ces données sur de longues périodes, et il est souvent souhaitable que ces données soient détruites lorsque la fonction qui les a allouées se termine. Autrement dit, il faudrait que les pointeurs détruisent automatiquement les données qu'ils référencent lorsqu'ils sont eux-mêmes détruits.

La bibliothèque standard C++ fournit à cet effet une classe d'encapsulation des pointeurs, qui permet d'obtenir ces fonctionnalités. Cette classe se nomme auto_ptr, en raison du fait que ses instances sont utilisées comme des pointeurs de données dont la portée est la même que celle des variables automatiques. La déclaration de cette classe est réalisée comme suit dans l'en-tête `memory` :

```
template <class T>
class auto_ptr
{
public:
    typedef T element_type;
    explicit auto_ptr(T *pointeur = 0) throw();
    auto_ptr(const auto_ptr &source) throw();
    template <class U>
    auto_ptr(const auto_ptr<U> &source) throw();
```

```
    ~auto_ptr() throw();

    auto_ptr &operator=(const auto_ptr &source) throw();
    template <class U>
    auto_ptr &operator=(const auto_ptr<U> &source) throw();

    T &operator*() const throw();
    T *operator->() const throw();
    T *get() const throw();
    T *release() const throw();
};
```

Cette classe permet de construire un objet contrôlant un pointeur sur un autre objet alloué dynamiquement avec l'opérateur `new`. Lorsqu'il est détruit, l'objet référencé est automatiquement détruit par un appel à l'opérateur `delete`. Cette classe utilise donc une sémantique de propriété stricte de l'objet contenu, puisque le pointeur ainsi contrôlé ne doit être détruit qu'une seule fois.

Cela implique plusieurs remarques. Premièrement, il y a nécessairement un transfert de propriété du pointeur encapsulé lors des opérations de copie et d'affectation. Deuxièmement, toute opération susceptible de provoquer la perte du pointeur encapsulé provoque sa destruction automatiquement. C'est notamment le cas lorsqu'une affectation d'une autre valeur est faite sur un auto_ptr contenant déjà un pointeur valide. Enfin, il ne faut jamais détruire soi-même l'objet pointé une fois que l'on a affecté un pointeur sur celui-ci à un auto_ptr.

Il est très simple d'utiliser les pointeurs automatiques. En effet, il suffit de les initialiser à leur construction avec la valeur du pointeur sur l'objet alloué dynamiquement. Dès lors, il est possible d'utiliser l'auto_ptr comme le pointeur original, puisqu'il définit les opérateurs '`*`' et '`->`'.

Les auto_ptr sont souvent utilisés en tant que variable automatique dans les sections de code susceptible de lancer des exceptions, puisque la remontée des exceptions détruit les variables automatiques. Il n'est donc plus nécessaire de traiter ces exceptions et de détruire manuellement les objets alloués dynamiquement avant de relancer l'exception.

Exemple 14-15. Utilisation des pointeurs automatiques

```
#include <iostream>
#include <memory>

using namespace std;

class A
{
public:
    A()
    {
        cout << "Constructeur" << endl;
    }

    ~A()
    {
        cout << "Destructeur" << endl;
    }
};

// Fonction susceptible de lancer une exception :
```

```
void f()
    // Alloue dynamiquement un objet :
    auto_ptr<A> p(new A);
    // Lance une exception, en laissant au pointeur
    // automatique le soin de détruire l'objet alloué :
    throw 2;
}

int main(void)
{
    try
    {
        f();
    }
    catch (...)
    {
    }
    return 0;
}
```

Note : On prendra bien garde au fait que la copie d'un auto_ptr dans un autre effectue un transfert de propriété. Cela peut provoquer des surprises, notamment si l'on utilise des paramètres de fonctions de type auto_ptr (chose expressément déconseillée). En effet, il y aura systématiquement transfert de propriété de l'objet lors de l'appel de la fonction, et c'est donc la fonction appelée qui en aura la responsabilité. Si elle ne fait aucun traitement spécial, l'objet sera détruit avec le paramètre de la fonction, lorsque l'exécution du programme en sortira ! Inutile de dire que la fonction appelante risque d'avoir des petits problèmes... Pour éviter ce genre de problèmes, il est plutôt conseillé de passer les auto_ptr par référence constante plutôt que par valeur dans les appels de fonctions.

Un autre piège classique est d'initialiser un auto_ptr avec l'adresse d'un objet qui n'a pas été alloué dynamiquement. Il est facile de faire cette confusion, car on ne peut a priori pas dire si un pointeur pointe sur un objet alloué dynamiquement ou non. Quoi qu'il en soit, si vous faites cette erreur, un appel à `delete` sera fait avec un paramètre incorrect lors de la destruction du pointeur automatique et le programme plantera.

Enfin, sachez que les pointeurs automatiques n'utilisent que l'opérateur `delete` pour détruire les objets qu'ils encapsulent, jamais l'opérateur `delete[]`. Par conséquent, les pointeurs automatiques ne devront jamais être initialisés avec des pointeurs obtenus lors d'une allocation dynamique avec l'opérateur `new[]` ou avec la fonction `malloc` de la bibliothèque C.

Il est possible de récupérer la valeur du pointeur pris en charge par un pointeur automatique simplement, grâce à la méthode `get`. Cela permet de travailler avec le pointeur original, cependant, il ne faut jamais oublier que c'est le pointeur automatique qui en a toujours la propriété. Il ne faut donc jamais appeler `delete` sur le pointeur obtenu.

En revanche, si l'on veut sortir le pointeur d'un auto_ptr, et forcer celui-ci à en abandonner la propriété, on peut utiliser la méthode `release`. Cette méthode renvoie elle-aussi le pointeur sur l'objet que l'auto_ptr contenait, mais libère également la référence sur l'objet pointé au sein de l'auto_ptr. Ainsi, la destruction du pointeur automatique ne provoquera plus la destruction de l'objet pointé et il faudra à nouveau prendre en charge cette destruction soi-même.

Exemple 14-16. Sortie d'un pointeur d'un auto_ptr

```
#include <iostream>
#include <memory>

using namespace std;

class A
{
public:
    A()
    {
        cout << "Constructeur" << endl;
    }

    ~A()
    {
        cout << "Destructeur" << endl;
    }
};

A *f(void)
{
    cout << "Construcion de l'objet" << endl;
    auto_ptr<A> p(new A);
    cout << "Extraction du pointeur" << endl;
    return p.release();
}

int main(void)
{
    A *pA = f();
    cout << "Destruction de l'objet" << endl;
    delete pA;
    return 0;
}
```

14.2.2. Les paires

Outre les pointeurs automatiques, la bibliothèque standard C++ définit une autre classe utilitaire qui permet quant à elle de stocker un couple de valeurs dans un même objet. Cette classe, la classe `template` pair, est en particulier très utilisée dans l'implémentation de certains conteneurs de la bibliothèque.

La déclaration de la classe `template` pair est la suivante dans l'en-tête `utility` :

```
template <class T1, class T2>
struct pair
{
    typedef T1 first_type;
    typedef T2 second_type;

    T1 first;
    T2 second;

    pair();
```

```
    pair(const T1 &, const T2 &);
    template <class U1, class U2>
    pair(const pair<U1, U2> &);
};

template <class T1, class T2>
bool operator==(const pair<T1, T2> &, const pair<T1, T2> &);

template <class T1, class T2>
bool operator<(const pair<T1, T2> &, const pair<T1, T2> &);

template <class T1, class T2>
pair<T1, T2> make_pair(const T1 &, const T2 &);
```

Comme cette déclaration le montre, l'utilisation de la classe pair est extrêmement simple. La construction d'une paire se fait soit en fournissant le couple de valeurs devant être stocké dans la paire, soit en appelant la fonction `make_pair`. La récupération des deux composantes d'une paire se fait simplement en accédant aux données membres publiques *`first`* et *`second`*.

Exemple 14-17. Utilisation des paires

```
#include <iostream>
#include <utility>

using namespace std;

int main(void)
{
    // Construit une paire associant un entier
    // à un flottant :
    pair<int, double> p1(5, 7.38), p2;
    // Initialise p2 avec make_pair :
    p2 = make_pair(9, 3.14);
    // Affiche les deux paires :
    cout << "p1 = (" << p1.first << ", "
        << p1.second << ")" << endl;
    cout << "p2 = (" << p2.first << ", "
        << p2.second << ")" << endl;
    return 0;
}
```

La classe `template` pair dispose également d'opérateurs de comparaison qui utilisent l'ordre lexicographique induit par les valeurs de ses deux éléments. Deux paires sont donc égales si et seulement si leurs couples de valeurs sont égaux membre à membre, et une paire est inférieure à l'autre si la première valeur de la première paire est inférieure à la valeur correspondante de la deuxième paire, ou, si elles sont égales, la deuxième valeur de la première paire est inférieure à la deuxième valeur de la deuxième paire.

14.3. Les types numériques

En plus des types d'intérêt général vus dans les sections précédentes, la bibliothèque standard fournit des types de données plus spécialisés dans les calculs numériques ou mathématiques. Ces types de données permettent d'effectuer des calculs sur les nombres complexes, ainsi que des calculs parallèles sur des tableaux de valeurs.

14.3.1. Les complexes

Les types de base du langage C++ fournissent une approximation relativement fiable des différents domaines de nombres mathématiques. Par exemple, le type int permet de représenter une plage de valeurs limitée des entiers relatifs, mais suffisamment large toutefois pour permettre d'effectuer la plupart des calculs intervenant dans la vie réelle. De même, les types des nombres à virgule flottante fournissent une approximation relativement satisfaisante des nombres réels des mathématiques. L'approximation cette fois porte non seulement sur la plage de valeur accessible, mais également sur la précision des nombres.

> **Note :** On prendra bien conscience du fait que les types du langage ne représentent effectivement que des approximations, car les ordinateurs sont des machines limitées en mémoire et en capacité de représentation du monde réel. Il faut donc toujours penser aux éventuels cas de débordements et erreurs de représentation des nombres, surtout en ce qui concerne les nombres réels. Les bogues les plus graves (en terme de pertes matérielles ou humaines) sont souvent dûs à de tels débordements, qui sont inhérents aux techniques utilisées par l'informatique (même avec des langages plus « sûrs » que le C++).

Il existe en mathématiques un autre type de nombres, qui n'ont pas de représentation physique immédiate pour le commun des mortels, mais qui permettent souvent de simplifier beaucoup certains calculs : les *nombres complexes*. Ces nombres étendent en effet le domaine des nombres accessibles et permettent de poursuivre les calculs qui n'étaient pas réalisables avec les nombres réels seulement, en s'affranchissant des contraintes imposées sur les solutions des équations algébriques. Les nombres complexes sont donc d'une très grande utilité dans toute l'algèbre, et en particulier dans les calculs matriciels où ils prennent une place prédominante. Les nombres complexes permettent également de simplifier sérieusement les calculs trigonométriques, les calculs de signaux en électricité et les calculs en mécanique quantique. Le plus intéressant avec ces nombres est sans doute le fait que même si les résultats intermédiaires que l'on trouve avec eux n'ont pas de signification réelle, les résultats finaux, eux, peuvent en avoir une et n'auraient pas été trouvés aussi facilement en conservant toutes les contraintes imposées par les nombres réels.

Afin de simplifier la vie des programmeurs qui ont besoin de manipuler des nombres complexes, la bibliothèque standard C++ définit la classe `template` complex, qui permet de les représenter et d'effectuer les principales opérations mathématiques dessus. Si l'utilisation de la classe complex en soi ne pose aucun problème particulier, il peut être utile de donner une description sommaire de ce qu'est un nombre complexe pour les néophytes en mathématiques. Toutefois, cette description n'est pas destinée aux personnes n'ayant aucune connaissance en mathématiques (si tant est qu'un programmeur puisse être dans ce cas...). Si vous ne la comprenez pas, c'est sans doute que vous n'avez aucunement besoin des nombres complexes et vous pouvez donc passer cette section sans crainte.

14.3.1.1. Définition et principales propriétés des nombres complexes

Il n'est pas compliqué de se représenter ce que signifie un nombre réel puisqu'on les utilise couramment dans la vie courante. La méthode la plus simple est d'imaginer une règle graduée où chaque position est donnée par un nombre réel par rapport à l'origine. Ce nombre indique le nombre de fois que l'unité de distance doit être répétée depuis l'origine pour arriver à cette position.

Pour se représenter la valeur d'un nombre complexe, il faut utiliser une dimension supplémentaire. En fait, tout nombre complexe peut être exprimé avec deux valeurs réelles : la *partie réelle* du complexe, et sa *partie imaginaire*. Plusieurs notations existent pour représenter les nombres complexes à partir de ces deux parties. La plus courante est de donner la partie réelle et la partie imaginaire entre parenthèses, séparées par une virgule :

```
(réelle, imaginaire)
```

où `réelle` est la valeur de la partie réelle, et `imaginaire` la valeur de la partie imaginaire. Il est également très courant en France de noter les deux parties directement en les séparant d'un signe d'addition et en accolant le caractère '`i`' (pour « imaginaire ») à la partie imaginaire :

```
réelle + imaginaire i
```

L'exemple suivant vous présente quelques nombres complexes :

```
7.56
3i
5+7i
```

Vous constaterez que les nombres réels peuvent parfaitement être représentés par les nombres complexes, puisqu'il suffit simplement d'utiliser une partie imaginaire nulle.

Les opérations algébriques classiques ont été définies sur les nombres complexes. Les additions et soustractions se font membre à membre, partie réelle avec partie réelle et partie imaginaire avec partie imaginaire. En revanche, la multiplication est un peu plus complexe, car elle se base sur la propriété fondamentale que le carré de l'unité de la partie imaginaire vaut `-1`. Autrement dit, le symbole `i` de la notation précédente dispose de la propriété fondamentale suivante : $i^2=-1$. Il s'agit en quelque sorte d'une racine carrée de `-1` (la racine carrée des nombres négatifs n'ayant pas de sens, puisqu'un carré est normalement toujours positif, on comprend la qualification d'« imaginaire » des nombres complexes). À partir de cette règle de base, et en conservant les règles d'associativité des opérateurs, on peut définir le produit de deux nombres complexes comme suit :

```
(a,b) * (c,d) = (ac - bd, ad + bc)
```

Enfin, la division se définit toujours comme l'opération inverse de la multiplication, c'est-à-dire l'opération qui trouve le nombre qui, multiplié par le diviseur, redonne le dividende. Chaque nombre complexe non nul dispose d'un inverse, qui est le résultat de la division de `1` par ce nombre. On peut montrer facilement que l'inverse d'un nombre complexe est défini comme suit :

$$1/(a,b) = (a / (a^2 + b^2), -b / (a^2 + b^2))$$

À partir de l'inverse, il est simple de calculer une division quelconque.

Comme il l'a été dit plus haut, les nombres complexes peuvent être représentés en utilisant une dimension supplémentaire. Ainsi, si on définit un repère dans le plan, dont l'axe des abscisses est associé à la partie réelle des nombres complexes et l'axe des ordonnées à la partie imaginaire, à tout nombre complexe est associé un point du plan. On appelle alors ce plan le *plan complexe*. La définition des complexes donnée ici correspond donc à un système de coordonnées cartésiennes du plan complexe, et chaque nombre complexe dispose de ses propres coordonnées.

En mathématiques, il est également courant d'utiliser un autre système de coordonnées : le *système de coordonnées polaires*. Dans ce système, chaque point du plan est identifié non plus par les coordonnées de ses projections orthogonales sur les axes du repère, mais par sa distance à l'origine et par l'angle que la droite qui rejoint l'origine au point fait avec l'axe des abscisses. Ces deux nombres sont couramment notés respectivement avec les lettres grecques rho et theta. La dénomination de coordonnées polaires provient du fait que l'origine du repère joue le rôle d'un *pôle* par rapport auquel on situe le point dans le plan.

Il est donc évident que les nombres complexes peuvent également être représentés par leurs coordonnées polaires. On appelle généralement la distance à l'origine la *norme* du nombre complexe, et l'angle qu'il fait avec l'axe des abscisses son *argument*. Faites bien attention à ce terme, il ne représente pas un argument d'une fonction ou quoi que ce soit qui se rapporte à la programmation.

La plupart des fonctions mathématiques classiques ont été définies sur les nombres complexes, parfois en restreignant leur domaine de validité. Ainsi, il est possible de calculer un sinus, un cosinus, une exponentielle, etc. pour les nombres complexes. Il est bien entendu hors de question de définir rigoureusement, ni même de présenter succinctement ces fonctions dans ce document. Cependant, il est bon de savoir qu'on ne peut pas définir une relation d'ordre sur les nombres complexes. Autrement dit, on ne peut pas faire d'autre comparaison que l'égalité entre deux nombres complexes (essayez de comparer les nombres complexes situés sur un cercle centré à l'origine dans le plan complexe pour vous en rendre compte).

14.3.1.2. La classe complex

La classe `template` complex est définie dans l'en-tête `complex` de la bibliothèque standard. Cette classe peut être instanciée pour l'un quelconque des trois types de nombre à virgule flottante du langage : float, double ou long double. Elle permet d'effectuer les principales opérations définies sur les nombres complexes, comme les additions, soustractions, multiplications, division, mais également des opérations spécifiques aux nombres complexes, comme la détermination de leur argument ou de leur norme. Enfin, l'en-tête `complex` contient des surcharges des fonctions mathématiques standards, telles que les fonctions trigonométriques, la racine carrée, les puissances et exponentielles, ainsi que les logarithmes (définis sur le plan complexe auquel l'axe des abscisses négatives a été ôté).

La construction d'un complexe ne pose aucun problème en soi. La classe complex dispose d'un constructeur par défaut, d'un constructeur de copie et d'un constructeur prenant en paramètre la partie réelle et la partie imaginaire du nombre :

```
 #include <iostream>
#include <complex>

using namespace std;

int main(void)
{
    complex<double> c(2,3);
    cout << c << endl;
    return 0;
}
```

L'exemple précédent présente également l'opérateur de sortie sur les flux standards, qui formate un nombre complexe en utilisant la notation `(réel,imaginaire)`. Il existe également une surcharge de l'opérateur d'entrée pour le flux d'entrée :

```
#include <iostream>
#include <complex>

using namespace std;

int main(void)
{
    complex<double> c;
    cin >> c;
    cout << "Vous avez saisi : " << c << endl;
    return 0;
}
```

> **Note :** Malheureusement, cette notation pose des problèmes avec la locale française, puisque nous utilisons des virgules pour séparer la partie entière de la partie décimale des nombres à virgules. Lorsque l'un des deux nombres flottants est un entier, il est impossible de déterminer où se trouve la virgule séparant la partie entière de la partie imaginaire du nombre complexe. Une première solution est de modifier le formatage des nombres réels pour que les chiffres après la virgule soient toujours affichés, même s'ils sont nuls. Cependant, il faut également imposer que les saisies des nombres soient également toujours effectués avec des nombres à virgules, ce qui est sujet à erreur et invérifiable. Il est donc recommandé de n'utiliser que la locale de la bibliothèque C lorsqu'on fait un programme utilisant les nombres complexes.

Il n'existe pas de constructeur permettant de créer un nombre complexe à partir de ses coordonnées polaires. En revanche, la fonction `polar` permet d'en construire un. Cette fonction prend en paramètre la norme du complexe à construire ainsi que son argument. Elle renvoie le nombre complexe nouvellement construit.

La partie imaginaire et la partie réelle d'un nombre complexe peuvent être récupérées à tout instant à l'aide des méthodes `real` et `imag` de la classe `template` complex. Il est également possible d'utiliser les fonctions `template real` et `imag`, qui prennent toutes deux le nombre complexe dont il faut calculer la partie réelle et la partie imaginaire. De même, la norme d'un nombre complexe est retournée par la fonction `abs`, et son argument peut être obtenu avec la fonction `arg`.

Bien entendu, les opérations classiques sur les complexes se font directement, comme s'il s'agissait d'un type prédéfini du langage :

```
#include <iostream>
#include <complex>

using namespace std;

int main(void)
{
    complex<double> c1(2.23, 3.56);
    complex<double> c2(5, 5);
    complex<double> c = c1+c2;
    c = c/(c1-c2);
    cout << c << endl;
    return 0;
}
```

Les fonctions spécifiques permettant de manipuler les complexes et de leur appliquer les opérations qui leurs sont propres sont récapitulées dans le tableau suivant :

Tableau 14-2. Fonctions spécifiques aux complexes

Fonction	Description
`real`	Retourne la partie réelle du nombre complexe.
`imag`	Retourne la partie imaginaire du nombre complexe.
`abs`	Retourne la norme du nombre nombre complexe, c'est-à-dire sa distance à l'origine.
`arg`	Retourne l'argument du nombre complexe.
`norm`	Retourne le carré de la norme du nombre complexe. Attention, cette fonction porte mal son nom, puisque la vraie norme est retournée par la surcharge de la fonction `abs` pour les nombres complexes. Cette incohérence provient de l'interprétation différente de celle des Français que font les Anglo-saxons de la notion de norme.
`conj`	Retourne le nombre complexe conjugué du nombre complexe fourni en argument. Le nombre conjugué d'un nombre complexe est son symétrique par rapport à l'axe des abscisses dans le plan complexe, c'est-à-dire qu'il dispose de la même partie réelle, mais que sa partie imaginaire est opposée à celle du nombre complexe original (cela revient également à dire que l'argument du conjugué est l'opposé de l'argument du complexe original). Le produit d'un nombre complexe et de son conjugué donne le carré de sa norme.
`polar`	Permet de construire un nombre complexe à partir de ses coordonnées polaires.

Exemple 14-18. Manipulation des nombres complexes

```
#include <iostream>
#include <complex>

using namespace std;

int main(void)
{
    // Crée un nombre complexe :
    complex<double> c(2,3);
    // Détermine son argument et sa norme :
    double Arg = arg(c);
    double Norm = abs(c);
    // Construit le nombre complexe conjugué :
    complex<double> co = polar(Norm, -Arg);
    // Affiche le carré de la norme du conjugué :
    cout << norm(co) << endl;
    // Calcule le carré ce cette norme par le produit
    // du complexe et de son conjugué :
    cout << real(c * conj(c)) << endl;
    return 0;
}
```

14.3.2. Les tableaux de valeurs

Comme il l'a été expliqué dans le Chapitre 1, les programmes classiques fonctionnent toujours sur le même principe : ils travaillent sur des données qu'ils reçoivent en entrée et produisent des résultats en sortie. Ce mode de fonctionnement convient dans la grande majorité des cas, et en fait les programmes que l'on appelle couramment les « filtres » en sont une des applications principales. Un *filtre* n'est rien d'autre qu'un programme permettant, comme son nom l'indique, de filtrer les données reçues en entrée selon un critère particulier et de ne fournir en sortie que les données qui satisfont ce critère. Certains filtres plus évolués peuvent même modifier les données à la volée ou les traduire dans un autre format. Les filtres sont très souvent utilisés avec les mécanismes de redirection des systèmes qui les supportent afin d'exécuter des traitements complexes sur les flux de données à partir de filtres simples, en injectant les résultats des uns dans le flux d'entrée des autres.

Cependant, ce modèle a une limite pratique en terme de performances, car il nécessite un traitement séquentiel des données. La vitesse d'exécution d'un programme conçu selon ce modèle est donc directement lié à la vitesse d'exécution des instructions, donc à la vitesse du processeur de la machine utilisée. Lorsqu'un haut niveau de performance doit être atteint, plusieurs solutions sont disponibles. Dans la pratique, on distingue trois solutions classiques.

La première solution consiste simplement, pour augmenter la puissance d'une machine, à augmenter celle du processeur. Cela se traduit souvent par une augmentation de la fréquence de ce processeur, technique que tout le monde connaît. Les avantages de cette solution sont évidents : tous les programmes bénéficient directement de l'augmentation de la puissance du processeur et n'ont pas à être modifiés. En revanche, cette technique atteindra un jour ou un autre ses limites en termes de coûts de fabrication et de moyens techniques à mettre en œuvre pour produire les processeurs.

La deuxième solution est d'augmenter le nombre de processeurs de la machine. Cette solution est très simple, mais suppose que les programmes soient capables d'effectuer plusieurs calculs indépendants simultanément. En particulier, les traitements à effectuer doivent être suffisamment indépendants et ne pas à avoir à attendre les données produites par les autres afin de pouvoir réellement être exécutés en parallèle. On quitte donc le modèle séquentiel, pour entrer dans un modèle de traitement où chaque processeur travaille en parallèle (modèle « MIMD », abréviation de l'anglais « Multiple Instruction Multiple Data »). Cette technique est également souvent appelée le *parallélisme de traitement*. Malheureusement, pour un unique processus purement séquentiel, cette technique ne convient pas, puisque de toutes façons, les opérations à exécuter ne le seront que par un seul processeur.

Enfin, il existe une technique mixte, qui consiste à paralléliser les données. Les mêmes opérations d'un programme séquentiel sont alors exécutées sur un grand nombre de données similaires. Les données sont donc traitées par blocs, par un unique algorithme : il s'agit du *parallélisme de données* (« SIMD » en anglais, abréviation de « Single Instruction Multiple Data »). Cette solution est celle mise en œuvre dans les processeurs modernes qui disposent de jeux d'instructions spécialisées permettant d'effectuer des calculs sur plusieurs données simultanément (MMX, 3DNow et SSE pour les processeurs de type x86 par exemple). Bien entendu, cette technique suppose que le programme ait effectivement à traiter des données semblables de manière similaire. Cette contrainte peut paraître très forte, mais, en pratique, les situations les plus consommatrices de ressources sont justement celles qui nécessite la répétition d'un même calcul sur plusieurs données. On citera par exemple tous les algorithmes de traitement de données multimédia, dont les algorithmes de compression, de transformation et de combinaison.

Si l'augmentation des performances des processeurs apporte un gain directement observable sur tous les programmes, ce n'est pas le cas pour les techniques de parallélisation. Le parallélisme de traitement est généralement accessible au niveau système, par l'intermédiaire du multitâche et de la programmation multithreadée. Il faut donc écrire les programmes de telle sorte à bénéficier de ce parallélisme de traitement, à l'aide des fonctions spécifique au système d'exploitation. De même, le parallélisme de données nécessite la définition de types de données complexes, capables de représen-

ter les blocs de données sur lesquels le programme doit travailler. Ces blocs de données sont couramment gérés comme des vecteurs ou des matrices, c'est-à-dire, en général, comme des tableaux de nombres. Le programme doit donc utiliser ces types spécifiques pour accéder à toutes les ressources de la machine. Cela nécessite un support de la part du langage de programmation.

Chaque environnement de développement est susceptible de fournir les types de données permettant d'effectuer des traitements SIMD. Cependant, ces types dépendent de l'environnement utilisé et encore plus de la plateforme utilisée. La bibliothèque standard C++ permet d'éviter ces écueils, car elle définit un type de donnée permettant de traiter des tableaux unidimensionnels d'objets, en assurant que les mécanismes d'optimisation propre aux plates-formes matérielles et aux compilateurs seront effectivement utilisés : les valarray.

14.3.2.1. Fonctionnalités de base des valarray

La classe valarray est une classe `template` capable de stocker un tableau de valeurs de son type `template`. Il est possible de l'instancier pour tous les types de données pour lesquels les opérations définies sur la classe valarray sont elles-mêmes définies. La bibliothèque standard C++ garantit que la classe valarray est écrite de telle sorte que tous les mécanismes d'optimisation des compilateurs pourront être appliqués sur elle, afin d'obtenir des performances optimales. De plus, chaque implémentation est libre d'utiliser les possibilités de calcul parallèle disponible sur chaque plateforme, du moins pour les types pour lesquels ces fonctionnalités sont présentes. Par exemple, la classe valarray instanciée pour le type float peut utiliser les instructions spécifiques de calcul sur les nombres flottants du processeur si elles sont disponibles. Toutefois, la norme n'impose aucune contrainte à ce niveau, et la manière dont la classe valarray est implémentée reste à la discrétion de chaque fournisseur.

La classe valarray fournit toutes les fonctionnalités nécessaires à la construction des tableaux de valeurs, à leur initialisation, ainsi qu'à leur manipulation. Elle est déclarée comme suit dans l'en-tête `valarray` :

```
// Déclaration des classes de sélection de sous-tableau :
class slice;
class gslice;

// Déclaration de la classe valarray :
template <class T>
class valarray
{
public:
    // Types des données :
    typedef T value_type;

    // Constructeurs et destructeurs :
    valarray();
    explicit valarray(size_t taille);
    valarray(const T &valeur, size_t taille);
    valarray(const T *tableau, size_t taille);
    valarray(const valarray &source);
    valarray(const mask_array<T> &source);
    valarray(const indirect_array<T> &source);
    valarray(const slice_array<T> &source);
    valarray(const gslice_array<T> &source);
    ~valarray();

    // Opérateurs d'affectation :
    valarray<T> &operator=(const T &valeur);
    valarray<T> &operator=(const valarray<T> &source);
```

```
valarray<T> &operator=(const mask_array<T> &source);
valarray<T> &operator=(const indirect_array<T> &source);
valarray<T> &operator=(const slice_array<T> &source);
valarray<T> &operator=(const gslice_array<T> &source);

// Opérateurs d'accès aux éléments :
T operator[](size_t indice) const;
T &operator[](size_t indice);

// Opérateurs de sélection de sous-ensemble du tableau :
valarray<T>       operator[](const valarray<bool> &masque) const;
mask_array<T>     operator[](const valarray<bool> &masque);
valarray<T>       operator[](const valarray<size_t> &indices) const;
indirect_array<T> operator[](const valarray<size_t> &indices);
valarray<T>       operator[](slice selecteur) const;
slice_array<T>    operator[](slice selecteur);
valarray<T>       operator[](const gslice &selecteur) const;
gslice_array<T>   operator[](const gslice &selecteur);

// Opérateurs unaires :
valarray<T> operator+() const;
valarray<T> operator-() const;
valarray<T> operator~() const;
valarray<T> operator!() const;

// Opérateurs d'affectation composée :
valarray<T> &operator*=(const T &valeur);
valarray<T> &operator*=(const valarray<T> &tableau);
valarray<T> &operator/=(const T &valeur);
valarray<T> &operator/=(const valarray<T> &tableau);
valarray<T> &operator%=(const T &valeur);
valarray<T> &operator%=(const valarray<T> &tableau);
valarray<T> &operator+=(const T &valeur);
valarray<T> &operator+=(const valarray<T> &tableau);
valarray<T> &operator-=(const T &valeur);
valarray<T> &operator-=(const valarray<T> &tableau);
valarray<T> &operator^=(const T &valeur);
valarray<T> &operator^=(const valarray<T> &tableau);
valarray<T> &operator&=(const T &valeur);
valarray<T> &operator&=(const valarray<T> &tableau);
valarray<T> &operator|=(const T &valeur);
valarray<T> &operator|=(const valarray<T> &tableau);
valarray<T> &operator<<=(const T &valeur);
valarray<T> &operator<<=(const valarray<T> &tableau);
valarray<T> &operator>>=(const T &valeur);
valarray<T> &operator>>=(const valarray<T> &tableau);

// Opérations spécifiques :
size_t size() const;
T sum() const;
T min() const;
T max() const;
valarray<T> shift(int) const;
valarray<T> cshift(int) const;
valarray<T> apply(T fonction(T)) const;
valarray<T> apply(T fonction(const T &)) const;
void resize(size_t taille, T initial=T());
```

```
};
```

Nous verrons dans la section suivante la signification des types slice, gslice, slice_array, gslice_array, mask_array et indirect_array.

Il existe plusieurs constructeurs permettant de créer et d'initialiser un tableau de valeurs. Le constructeur par défaut initialise un tableau de valeur vide. Les autres constructeurs permettent d'initialiser le tableau de valeur à partir d'une valeur d'initialisation pour tous les éléments du valarray, ou d'un autre tableau contenant les données à affecter aux éléments du valarray :

```
// Construit un valarray de doubles :
valarray<double> v1;

// Initialise un valarray de doubles explicitement :
double valeurs[] = {1.2, 3.14, 2.78, 1.414, 1.732};
valarray<double> v2(valeurs,
    sizeof(valeurs) / sizeof(double));

// Construit un valarray de 10 entiers initialisés à 3 :
valarray<int> v3(3, 10);
```

Vous pouvez constater que le deuxième argument des constructeurs qui permettent d'initialiser les valarray prennent un argument de type size_t, qui indique la taille du valarray. Une fois un valarray construit, il est possible de le redimensionner à l'aide de la méthode `resize`. Cette méthode prend en premier paramètre la nouvelle taille du valarray et la valeur à utiliser pour réinitialiser tous les éléments du valarray après redimensionnement. La valeur par défaut est celle fournie par le constructeur par défaut du type des données contenues dans le valarray. La taille courante d'un valarray peut être récupérée à tout moment grâce à la méthode `size`.

Exemple 14-19. Modification de la taille d'un valarray

```
#include <iostream>
#include <valarray>

using namespace std;

int main(void)
{
    // Création d'un valarray :
    valarray<double> v;
    cout << v.size() << endl;
    // Redimensionnement du valarray :
    v.resize(5, 3.14);
    cout << v.size() << endl;
    return 0;
}
```

Toutes les opérations classiques des mathématiques peuvent être appliquées sur un valarray pourvu qu'elles puissent l'être également sur le type des données contenues par ce tableau. La définition de ces opérations est très simple : l'opération du type de base est appliquée simplement à chaque élément contenu dans le tableau de valeurs.

La bibliothèque standard définit également les opérateurs binaires nécessaires pour effectuer les opérations binaires sur chaque élément des valarray. En fait, ces opérateurs sont classés en deux catégories, selon la nature de leurs arguments. Les opérateurs de la première catégorie permettent d'effectuer une opération entre deux valarray de même dimension, en appliquant cette opération membre à membre. Il s'agit donc réellement d'une opération vectorielle dans ce cas. En revanche, les opérateurs de la deuxième catégorie appliquent l'opération avec une même et unique valeur pour chaque donnée stockée dans le valarray.

Exemple 14-20. Opérations sur les valarray

```
#include <iostream>
#include <valarray>

using namespace std;

void affiche(const valarray<double> &v)
{
    size_t i;
    for (i=0; i<v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;
}

int main(void)
{
    // Construit deux valarray de doubles :
    double v1[] = {1.1, 2.2, 3.3};
    double v2[] = {5.3, 4.4, 3.5};
    valarray<double> vect1(v1, 3);
    valarray<double> vect2(v2, 3);
    valarray<double> res(3);
    // Effectue une somme membre à membre :
    res = vect1 + vect2;
    affiche(res);
    // Calcule le sinus des membres du premier valarray :
    res = sin(vect1);
    affiche(res);
    return 0;
}
```

Parmi les opérateurs binaires que l'on peut appliquer à un valarray, on trouve bien entendu les opérateurs de comparaison. Ces opérateurs, contrairement aux opérateurs de comparaison habituels, ne renvoient pas un booléen, mais plutôt un autre tableau de booléens. En effet, la comparaison de deux valarray a pour résultat le valarray des résultats des comparaisons membres à membres des deux valarray.

La classe valarray dispose de méthodes permettant d'effectuer diverses opérations spécifiques aux tableaux de valeurs. La méthode `sum` permet d'obtenir la somme de toutes les valeurs stockées dans le tableau de valeur. Les méthodes `shift` et `cshift` permettent, quant à elles, de construire un nouveau valarray dont les éléments sont les éléments du valarray auquel la méthode est appliquée, décalés ou permutés circulairement d'un certain nombre de positions. Le nombre de déplacements effectués est passé en paramètre à ces deux fonctions, les valeurs positives entraînant des déplacements vers la gauche et les valeurs négatives des déplacements vers la droite. Dans le cas des décalages les nouveaux éléments introduits pour remplacer ceux qui n'ont pas eux-mêmes de remplaçant prennent la valeur spécifiée par le constructeur par défaut du type utilisé.

Exemple 14-21. Décalages et rotations de valeurs

```
#include <iostream>
#include <valarray>

using namespace std;

void affiche(const valarray<double> &v)
{
    size_t i;
    for (i=0; i<v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;
}

int main(void)
{
    // Construit un valarray de doubles :
    double v1[] = {1.1, 2.2, 3.3, 4.4, 5.5};
    valarray<double> vect1(v1, 5);
    valarray<double> res(5);
    // Effectue un décalage à gauche de deux positions :
    res = vect1.shift(2);
    affiche(res);
    // Effectue une rotation de 2 positions vers la droite :
    res = vect1.cshift(-2);
    affiche(res);
    return 0;
}
```

Enfin, il existe deux méthodes `apply` permettant d'appliquer une fonction à chaque élément d'un valarray et de construire un nouveau valarray de même taille et contenant les résultats. Ces deux surcharges peuvent travailler respectivement avec des fonctions prenant en paramètre soit par valeur, soit par référence, l'objet sur lequel elles doivent être appliquées.

14.3.2.2. Sélection multiple des éléments d'un valarray

Les éléments d'un valarray peuvent être accédés à l'aide de l'opérateur d'accès aux éléments de tableau '`[]`'. La fonction `affiche` des exemples du paragraphe précédent utilise cette fonctionnalité pour en récupérer la valeur. Cependant, les valarray dispose de mécanismes plus sophistiqués pour manipuler les éléments des tableaux de valeur en groupe, afin de bénéficier de tous les mécanismes d'optimisation qui peuvent exister sur une plateforme donnée. Grâce à ces mécanismes, il est possible d'effectuer des opérations sur des parties seulement d'un valarray ou d'écrire de nouvelles valeurs dans certains de ses éléments seulement.

Pour effectuer ces sélections multiples, plusieurs techniques sont disponibles. Cependant, toutes ces techniques se basent sur le même principe, puisqu'elles permettent de filtrer les éléments du valarray pour n'en sélectionner qu'une partie seulement. Le résultat de ce filtrage peut être un nouveau valarray ou une autre classe pouvant être manipulée exactement de la même manière qu'un valarray.

En pratique, il existe quatre manières de sélectionner des éléments dans un tableau. Nous allons les détailler dans les sections suivantes.

14.3.2.2.1. Sélection par un masque

La manière la plus simple est d'utiliser un masque de booléens indiquant quels éléments doivent être sélectionnés ou non. Le masque de booléens doit obligatoirement être un valarray de même dimension que le valarray contenant les éléments à sélectionner. Chaque élément est donc sélectionné en fonction de la valeur du booléen correspondant dans le masque.

Une fois le masque construit, la sélection des éléments peut être réalisée simplement en fournissant ce masque à l'opérateur `[]` du valarray contenant les éléments à sélectionner. La valeur retournée par cet opérateur est alors une instance de la classe `template` mask_array, par l'intermédiaire de laquelle les éléments sélectionnés peuvent être manipulés. Pour les valarray constants cependant, la valeur retournée est un autre valarray, contenant une copie des éléments sélectionnés.

La classe mask_array fournit un nombre limité d'opérations. En fait, ses instances ne doivent être utilisées que pour effectuer des opérations simples sur les éléments du tableau sélectionné par le masque fourni à l'opérateur `[]`. Les opérations réalisables seront décrites dans la Section 14.3.2.2.4.

La sélection des éléments d'un tableau par l'intermédiaire d'un masque est utilisée couramment avec les opérateurs de comparaison des valarray, puisque ceux-ci renvoient justement un tel masque. Il est donc très facile d'effectuer des opérations sur les éléments d'un valarray qui vérifient une certaine condition.

Exemple 14-22. Sélection des éléments d'un valarray par un masque

```
#include <iostream>
#include <valarray>

using namespace std;

void affiche(const valarray<int> &v)
{
    size_t i;
    for (i=0; i<v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;
}

int main(void)
{
    // Construit un valarray d'entier :
    int valeurs[] = { 1, 5, 9, 4, 3, 7, 21, 32 };
    valarray<int> vi(valeurs,
        sizeof(valeurs) / sizeof(int));
    affiche(vi);
    // Multiplie par 2 tous les multiples de 3 :
    vi[(vi % 3)==0] *= valarray<int>(2, vi.size());
    affiche(vi);
    return 0;
}
```

14.3.2.2.2. Sélection par indexation explicite

La sélection des éléments d'un valarray par un masque de booléens est explicite et facile à utiliser, mais elle souffre de plusieurs défauts. Premièrement, il faut fournir un tableau de booléen de même dimension que le valarray source. Autrement dit, il faut fournir une valeur booléenne pour tous les

éléments du tableau, même pour ceux qui ne nous intéressent pas. Ensuite, les éléments sélectionnés apparaissent systématiquement dans le même ordre que celui qu'ils ont dans le valarray source.

La bibliothèque standard C++ fournit donc un autre mécanisme de sélection, toujours explicite, mais qui permet de faire une réindexation des éléments ainsi sélectionnés. Cette fois, il ne faut plus fournir un masque à l'opérateur `[]`, mais un valarray contenant directement les indices des éléments sélectionnés. Ces indices peuvent ne pas être dans l'ordre croissant, ce qui permet donc de réarranger l'ordre des éléments ainsi sélectionnés.

Exemple 14-23. Sélection des éléments d'un valarray par indexation

```
#include <iostream>
#include <valarray>

using namespace std;

void affiche(const valarray<int> &v)
{
    size_t i;
    for (i=0; i<v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;
}

int main(void)
{
    // Construit un valarray d'entier :
    int valeurs[] = { 1, 5, 9, 4, 3, 7, 21, 32 };
    valarray<int> vi(valeurs,
        sizeof(valeurs) / sizeof(int));
    affiche(vi);
    // Multiplie par 2 les éléments d'indices 2, 5 et 7 :
    size_t indices[] = {2, 5, 7};
    valarray<size_t> ind(indices,
        sizeof(indices) / sizeof(size_t));
    vi[ind] *= valarray<int>(2, ind.size());
    affiche(vi);
    return 0;
}
```

La valeur retournée par l'opérateur de sélection sur les valarray non constants est cette fois du type indirect_array. Comme pour la classe mask_array, les opérations réalisables par l'intermédiaire de cette classe sont limitées et doivent servir uniquement à modifier les éléments sélectionnés dans le valarray source.

14.3.2.2.3. Sélection par indexation implicite

Dans beaucoup de situations, les indices des éléments sélectionnés suivent un motif régulier et il n'est pas toujours pratique de spécifier ce motif explicitement. La méthode de sélection précédente n'est dans ce cas pas très pratique et il est alors préférable de sélectionner les éléments par un jeu d'indices décrits de manière implicite. La bibliothèque fournit à cet effet deux classes utilitaires permettant de décrire des jeux d'indices plus ou moins complexes : la classe slice et la classe gslice.

Ces deux classes définissent les indices des éléments à sélectionner à l'aide de plusieurs variables pouvant prendre un certain nombre de valeurs espacées par un pas d'incrémentation fixe. La définition des indices consiste donc simplement à donner la valeur de départ de l'indice de sélection, le nombre

de valeurs à générer pour chaque variable et le pas qui sépare ces valeurs. Les variables de contrôle commencent toutes leur itération à partir de la valeur nulle et prennent comme valeurs successives les multiples du pas qu'elles utilisent.

> **Note :** En réalité, la classe slice est un cas particulier de la classe gslice qui n'utilise qu'une seule variable de contrôle pour définir les indices. Les slice ne sont donc rien d'autre que des gslice unidimensionnels.
>
> Le terme de gslice provient de l'anglais « Generalized Slice », qui signifie bien que les gslice sont des slice étendues à plusieurs dimensions.

La classe slice est relativement facile à utiliser, puisqu'il suffit de spécifier la valeur de départ de l'indice, le nombre de valeurs à générer et le pas qui doit les séparer. Elle est déclarée comme suit dans l'en-tête `valarray` :

```
class slice
{
public:
    slice();
    slice(size_t debut, size_t nombre, size_t pas);

    // Accesseurs :
    size_t start() const;
    size_t size() const;
    size_t stride() const;
};
```

Exemple 14-24. Sélection par indexation implicite

```
#include <iostream>
#include <valarray>

using namespace std;

void affiche(const valarray<int> &v)
{
    size_t i;
    for (i=0; i<v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;
}

int main(void)
{
    // Construit un valarray d'entier :
    int valeurs[] = { 1, 5, 9, 4, 3, 7, 21, 32 };
    valarray<int> vi(valeurs, 8);
    affiche(vi);
    // Multiplie par 2 un élément sur 3 à partir du deuxième :
    slice sel(1, 3, 3);
    vi[sel] *= valarray<int>(2, vi.size());
    affiche(vi);
    // Multiplie par 2 un élément sur 3 à partir du deuxième :
    slice sel(1, 3, 3);
```

```
    vi[sel] *= valarray<int>(2, vi.size());
    affiche(vi);
    return 0;
}
```

La classe gslice est en revanche un peu plus difficile d'emploi puisqu'il faut donner le nombre de valeurs et le pas pour chaque variable de contrôle. Le constructeur utilisé prend donc en deuxième et troisième paramètres non plus deux valeurs de type size_t, mais deux valarray de size_t. La déclaration de la classe gslice est donc la suivante :

```
class gslice
{
public:
    gslice();
    gslice(size_t debut,
        const valarray<size_t> nombres,
        const valarray<size_t> pas);

    // Accesseurs :
    size_t start() const;
    valarray<size_t> size() const;
    valarray<size_t> stride() const;
};
```

Les deux valarray déterminant le nombre de valeurs des variables de contrôle et leurs pas doivent bien entendu avoir la même taille. L'ordre dans lequel les indices des éléments sélectionnés sont générés par la classe gslice est celui obtenu en faisant varier en premier les dernières variables caractérisées par les valarray fournis lors de sa construction. Par exemple, une classe gslice utilisant trois variables prenant respectivement 2, 3 et 5 valeurs et variant respectivement par pas de 3, 1 et 2 unités, en partant de l'indice 2, générera les indices suivants :

```
2, 4, 6, 8, 10,
3, 5, 7, 9, 11,
4, 6, 8, 10, 12,

5, 7, 9, 11, 13,
6, 8, 10, 12, 14,
7, 9, 11, 13, 15
```

La variable prenant cinq valeurs et variant de deux en deux est donc celle qui évolue le plus vite.

Comme vous pouvez le constater avec l'exemple précédent, un même indice peut apparaître plusieurs fois dans la série définie par une classe gslice. La bibliothèque standard C++ n'effectue aucun contrôle à ce niveau : il est donc du ressort du programmeur de bien faire attention à ce qu'il fait lorsqu'il manipule des jeux d'indices dégénérés.

Comme pour les autres techniques de sélection, la sélection d'éléments d'un valarray non constant par l'intermédiaire des classes slice et gslice retourne une instance d'une classe particulière permettant de prendre en charge les opérations de modification des éléments ainsi sélectionnés. Pour les sélections simples réalisées avec la classe slice, l'objet retourné est de type slice_array. Pour les sélections réalisées avec la classe gslice, le type utilisé est le type gslice_array.

14.3.2.2.4. Opérations réalisables sur les sélections multiples

Comme on l'a vu dans les sections précédentes, les sélections multiples réalisées sur des objets non constants retournent des instances des classes utilitaires mask_array, indexed_array, slice_array et gslice_array. Ces classes référencent les éléments ainsi sélectionnés dans le valarray source, permettant ainsi de les manipuler en groupe. Cependant, ce ne sont pas des valarray complets et, en fait, ils ne doivent être utilisés, de manière générale, que pour effectuer une opération d'affectation sur les éléments sélectionnés. Ces classes utilisent donc une interface restreinte de celle de la classe valarray, qui n'accepte que les opérateurs d'affectation sur les éléments qu'elles représentent.

Par exemple, la classe mask_array est déclarée comme suit dans l'en-tête `valarray` :

```
template <class T>
class mask_array
{
public:
    typedef T value_type;
    ~mask_array();

    // Opérateurs d'affectation et d'affectation composées :
    void operator=(const valarray<T> &) const;
    void operator*=(const valarray<T> &) const;
    void operator/=(const valarray<T> &) const;
    void operator%=(const valarray<T> &) const;
    void operator+=(const valarray<T> &) const;
    void operator-=(const valarray<T> &) const;
    void operator^=(const valarray<T> &) const;
    void operator&=(const valarray<T> &) const;
    void operator|=(const valarray<T> &) const;
    void operator<<=(const valarray<T> &) const;
    void operator>>=(const valarray<T> &) const;
    void operator=(const T &valeur);
};
```

Tous ces opérateurs permettent d'affecter aux éléments de la sélection représentés par cette classe les valeurs spécifiées par leur paramètre. En général, ces valeurs doivent être fournies sous la forme d'un valarray, mais il existe également une surcharge de l'opérateur d'affectation permettant de leur affecter à tous une même valeur.

> **Note :** Les sélections réalisées sur les valarray constants ne permettent bien entendu pas de modifier leurs éléments. Les objets retournés par l'opérateur `[]` lors des sélections multiples sur ces objets sont donc des valarray classiques contenant une copie des valeurs des éléments sélectionnés.

14.3.3. Les champs de bits

De tous les types de données qu'un programme peut avoir besoin de stocker, les booléens sont certainement l'un des plus importants. En effet, les programmes doivent souvent représenter des propriétés qui sont soit vraies, soit fausses. Après tout, la base du traitement de l'information telle qu'il est réalisé par les ordinateurs est le bit, ou chiffre binaire...

Il existe plusieurs manières de stocker des booléens dans un programme. La technique la plus simple est bien entendu d'utiliser le type C++ natif bool, qui ne peut prendre que les valeurs `true` et `false`. Les programmes plus vieux utilisaient généralement des entiers et des constantes prédéfinies ou encore une énumération. Malheureusement, toutes ces techniques souffrent du gros inconvénient que chaque information est stockée dans le type sous-jacent au type utilisé pour représenter les booléens et, dans la plupart des cas, ce type est un entier. Cela signifie que pour stocker un bit, il faut réserver un mot mémoire complet. Même en tenant compte du fait que la plupart des compilateurs C++ stockent les variables de type bool dans de simples octets, la déperdition reste dans un facteur 8. Bien entendu, cela n'est pas grave si l'on n'a que quelques bits à stocker, mais si le programme doit manipuler un grand nombre d'informations booléennes, cette technique est à proscrire.

Nous avons vu dans la Section 3.2.5 qu'il est possible de définir des champs de bits en attribuant un nombre de bits fixe à plusieurs identificateurs de type entier. Cette solution peut permettre d'économiser de la mémoire, mais reste malgré tout relativement limitée si un grand nombre de bits doit être manipulé. Afin de résoudre ce problème, la bibliothèque standard C++ fournit la classe `template` bitset qui, comme son nom l'indique, encapsule des champs de bits de tailles arbitraires. Le paramètre `template` est de type size_t et indique le nombre de bits que le champ de bits encapsulé contient.

> **Note :** Vous noterez que cela impose de connaître à la compilation la taille du champ de bits. Cela est regrettable et limite sérieusement l'intérêt de cette classe. Si vous devez manipuler des champs de bits de taille dynamique, vous devrez écrire vous-même une classe d'encapsulation dynamique des champs de bits.

La classe bitset est déclarée comme suit dans l'en-tête `bitset` :

```
template <size_t N>
class bitset
{
public:
    class reference;    // Classe permettant de manipuler les bits.

    // Les constructeurs :
    bitset();
    bitset(unsigned long val);
    template<class charT, class traits, class Allocator>
    explicit bitset(
        const basic_string<charT, traits, Allocator> &chaine,
        typename basic_string<charT, traits, Allocator>::size_type debut = 0,
        typename basic_string<charT, traits, Allocator>::size_type taille =
            basic_string<charT, traits, Allocator>::npos);

    // Les fonctions de conversion :
    unsigned long to_ulong() const;
    template <class charT, class traits, class Allocator>
        basic_string<charT, traits, Allocator> to_string() const;

    // Les opérateurs de manipulation :
    bitset<N> &operator&=(const bitset<N> &);
    bitset<N> &operator|=(const bitset<N> &);
    bitset<N> &operator^=(const bitset<N> &);
    bitset<N> &operator<<=(size_t pos);
    bitset<N> &operator>>=(size_t pos);
    bitset<N> operator<<(size_t pos) const;
```

```
    bitset<N> operator>>(size_t pos) const;
    bitset<N>  operator~() const;
    bitset<N> &set();
    bitset<N> &set(size_t pos, bool val = true);
    bitset<N> &reset();
    bitset<N> &reset(size_t pos);
    bitset<N> &flip();
    bitset<N> &flip(size_t pos);
    bool test(size_t pos) const;
    reference operator[](size_t pos);   // for b[i];

    // Les opérateurs de comparaison :
    bool operator==(const bitset<N> &rhs) const;
    bool operator!=(const bitset<N> &rhs) const;

    // Les fonctions de test :
    size_t count() const;
    size_t size() const;
    bool any() const;
    bool none() const;
};
```

La construction d'un champ de bits nécessite de connaître le nombre de bits que ce champ doit contenir afin d'instancier la classe `template` bitset. Les différents constructeurs permettent d'initialiser le champ de bits en affectant la valeur nulle à tous ses bits ou en les initialisant en fonction des paramètres du constructeur. Le deuxième constructeur affectera aux premiers bits du champ de bits les bits correspondant de l'entier de type unsigned long fourni en paramètre, et initialisera les autres bits du champ de bits à la valeur 0 si celui-ci contient plus de bits qu'un unsigned long. Le troisième constructeur initialise le champ de bits à partir de sa représentation sous forme de chaîne de caractères ne contenant que des '`0`' ou des '`1`'. Cette représentation doit être stockée dans la basic_string fournie en premier paramètre, à partir de la position `debut` et sur une longueur de `taille` caractères. Cette taille peut être inférieure à la taille du champ de bits. Dans ce cas, le constructeur considérera que les bits de poids fort sont tous nuls et initialisera les premiers bits du champ avec les valeurs lues dans la chaîne. Notez bien que les premiers caractères de la chaîne de caractères représentent les bits de poids fort, cette chaîne est donc parcourue en sens inverse lors de l'initialisation. Ce constructeur est susceptible de lancer une exception `out_of_range` si le paramètre `debut` est supérieur à la taille de la chaîne ou une exception `invalid_argument` si l'un des caractères utilisés est différent des caractères '`0`' ou '`1`'.

Comme vous pouvez le constater d'après la déclaration, la classe bitset fournit également des méthodes permettant d'effectuer les conversions inverses de celles effectuées par les constructeurs. La méthode `to_ulong` renvoie donc un entier de type unsigned long correspondant à la valeur des premiers bits du champ de bits, et la méthode `template to_string` renvoie une chaîne de caractères contenant la représentation du champ de bits sous la forme d'une suite de caractères '`0`' et '`1`'. La classe bitset fournit également des surcharges des opérateurs `operator<<` et `operator>>` pour les flux d'entrée / sortie de la bibliothèque standard.

Exemple 14-25. Utilisation d'un bitset

```
#include <iostream>
#include <bitset>
#include <string>

using namespace std;
```

```
int main(void)
{
    // Construit un champ de bits :
    string s("100110101");
    bitset<32> bs(s);
    // Affiche la valeur en hexadécimal de l'entier associé :
    cout << hex << showbase << bs.to_ulong() << endl;
    // Affiche la valeur sous forme de chaîne de caractères :
    string t;
    t = bs.to_string<string::value_type, string::traits_type,
        string::allocator_type>();
    cout << t << endl;
    // Utilise directement << sur le flux de sortie :
    cout << bs << endl;
    return 0;
}
```

> **Note :** La méthode `to_string` est une fonction `template` ne prenant pas de paramètres. Le compilateur ne peut donc pas réaliser une instanciation implicite lors de son appel. Par conséquent, vous devrez fournir la liste des paramètres `template` explicitement si vous désirez utiliser cette méthode. Il est généralement plus simple d'écrire la valeur du bitset dans un flux standard.
>
> Les modificateurs de format de flux `hex` et `showbase` ont pour but d'effectuer l'affichage des entiers sous forme hexadécimale. La personnalisation des flux d'entrée / sortie sera décrite en détail dans le Chapitre 15.

Les opérateurs de manipulation des champs de bits ne posent pas de problème particulier puisqu'ils ont la même sémantique que les opérateurs standards du langage, à ceci près qu'ils travaillent sur l'ensemble des bits du champ en même temps. Le seul opérateur qui demande quelques explications est l'opérateur d'accès unitaire aux bits du champ, à savoir l'opérateur `operator[]`. En effet, cet opérateur ne peut pas retourner une référence sur le bit désigné par son argument puisqu'il n'y a pas de type pour représenter les bits en C++. Par conséquent, la valeur retournée est en réalité une instance de la sous-classe reference de la classe bitset. Cette sous-classe encapsule l'accès individuel aux bits d'un champ de bits et permet de les utiliser exactement comme un booléen. En particulier, il est possible de faire des tests directement sur cette valeur ainsi que de lui affectuer une valeur booléenne. Enfin, la sous-classe reference dispose d'une méthode `flip` dont le rôle est d'inverser la valeur du bit auquel l'objet reference donne accès.

La classe `template` bitset dispose également de méthodes spécifiques permettant de manipuler les bits sans avoir recours à l'opérateur `operator[]`. Il s'agit des méthodes `test`, `set`, `reset` et `flip`. La première méthode permet de récupérer la valeur courante d'un des bits du champ de bits. Elle prend en paramètre le numéro de ce bit et renvoie un booléen valant `true` si le bit est à `1` et `false` sinon. La méthode `set` permet de réinitialiser le champ de bits complet en positionnant tous ses bits à 1 ou de fixer manuellement la valeur d'un bit particulier. La troisième méthode permet de réinitialiser le champ de bits en annulant tous ses bits ou d'annuler un bit spécifique. Enfin, la méthode `flip` permet d'inverser la valeur de tous les bits du champ ou d'inverser la valeur d'un bit spécifique. Les surcharges des méthodes qui travaillent sur un seul bit prennent toutes en premier paramètre la position du bit dans le champ de bits.

Exemple 14-26. Manipulation des bits d'un champ de bits

```
#include <iostream>
#include <string>
```

```
#include <bitset>

using namespace std;

int main(void)
{
    // Construit un champ de bits :
    string s("10011010");
    bitset<8> bs(s);
    cout << bs << endl;
    // Inverse le champ de bits :
    bs.flip();
    cout << bs << endl;
    // Fixe le bit de poids fort :
    bs.set(7, true);
    cout << bs << endl;
    // Annule le 7ème bit à l'aide d'une référence de bit :
    bs[6] = false;
    cout << bs << endl;
    // Anule le bit de poids faibe :
    bs.reset(0);
    cout << bs << endl;
    return 0;
}
```

Enfin, la classe bitset fournit quelques méthodes permettant d'effectuer des tests sur les champs de bits. Outre les opérateurs de comparaison classiques, elle fournit les méthodes `count`, `size`, `any` et `none`. La méthode `count` renvoie le nombre de bits positionnés à 1 dans le champ de bits. La méthode `size` renvoie quant à elle la taille du champ de bits, c'est-à-dire la valeur du paramètre `template` utilisée pour instancier la classe bitset. Enfin, les méthodes `any` et `none` renvoient `true` si un bit au moins du champ de bits est positionné ou s'ils sont tous nuls.

Chapitre 15. Les flux d'entrée / sortie

Nous avons vu dans la Section 7.12 un exemple d'application des classes de flux d'entrée / sortie de la bibliothèque pour les entrées / sorties standards des programmes. En réalité, ces classes de gestion des flux s'intègrent dans une hiérarchie complexe de classes permettant de manipuler les flux d'entrée / sortie et pas seulement pour les entrées / sorties standards.

En effet, afin de faciliter la manipulation des flux d'entrée / sortie, la bibliothèque standard C++ fournit tout un ensemble de classes `template`. Ces classes sont paramétrées par le type de base des caractères qu'elles manipulent. Bien entendu, les types de caractères les plus utilisés sont les type char et wchar_t, mais il est possible d'utiliser a priori n'importe quel autre type de donnée pour lequel une classe de traits char_traits est définie.

Ce chapitre a pour but de détailler cette hiérarchie de classes. Les principes de base et l'architecture générale des flux C++ seront donc abordés dans un premier temps, puis les classes de gestion des tampons seront traitées. Les classes génériques de gestion des flux d'entrée / sortie seront ensuite décrites, et ce sera enfin le tour des classes de gestion des flux orientés chaînes de caractères et des classes de gestion des flux orientés fichiers.

15.1. Notions de base et présentation générale

Les classes de la bibliothèque d'entrée / sortie de la bibliothèque standard se subdivisent en deux catégories distinctes.

La première catégorie regroupe les classes de gestion des tampons d'entrée / sortie. Ces classes sont au nombre de trois : la classe `template` basic_stringbuf, qui permet de réaliser des tampons pour les flux orientés chaînes de caractères, la classe `template` basic_filebuf, qui prend en charge les tampons pour les flux orientés fichiers, et leur classe de base commune, la classe `template` basic_streambuf. Le rôle de ces classes est principalement d'optimiser les entrées / sorties en intercalant des tampons d'entrée / sortie au sein même du programme. Ce sont principalement des classes utilitaires, qui sont utilisées en interne par les autres classes de la bibliothèque d'entrée / sortie.

La deuxième catégorie de classes est de loin la plus complexe, puisqu'il s'agit des classes de gestion des flux eux-mêmes. Toutes ces classes dérivent de la classe `template` basic_ios (elle-même dérivée de la classe de base ios_base, qui définit tous les types et les constantes utilisés par les classes de flux). La classe basic_ios fournit les fonctionnalités de base des classes de flux et, en particulier, elle gère le lien avec les tampons d'entrée / sortie utilisés par le flux. De cette classe de base dérivent des classes spécialisées respectivement pour les entrées ou pour les sorties. Ainsi, la classe `template` basic_istream prend en charge toutes les opérations des flux d'entrée et la classe basic_ostream toutes les opérations des flux de sortie. Enfin, la bibliothèque standard définit la classe `template` basic_iostream, qui regroupe toutes les fonctionnalités des classes basic_istream et basic_ostream et dont dérivent toutes les classes de gestion des flux mixtes.

Les classes basic_istream, basic_ostream et basic_iostream fournissent les fonctionnalités de base des flux d'entrée / sortie. Ce sont donc les classes utilisées pour implémenter les flux d'entrée / sortie standards du C++ `cin`, `cout`, `cerr` et `clog`, que l'on a brièvement présentés dans la Section 7.12. Cependant, ces classes ne prennent pas en charge toutes les spécificités des médias avec lesquels des flux plus complexes peuvent communiquer. Par conséquent, des classes dérivées, plus spécialisées, sont fournies par la bibliothèque standard. Ces classes prennent en charge les entrées / sorties sur fichier et les flux orientés chaînes de caractères.

La bibliothèque standard fournit donc deux jeux de classes spécialisées pour les entrées / sorties dans des fichiers et dans des chaînes de caractères. Pour chacune des classes de base basic_istream,

basic_ostream et basic_iostream il existe deux classes dérivées, une pour les fichiers, et une pour les chaînes de caractères. Par exemple, les classes `template` basic_ifstream et basic_istringstream dérivent de la classe basic_istream et prennent en charge respectivement les flux d'entrée à partir de fichiers et les flux d'entrée à partir de chaînes de caractères. De même, la bibliothèque standard définit les classes `template` basic_ofstream et basic_ostringstream, dérivées de la classe basic_ostream, pour les flux de sortie dans des fichiers ou dans des chaînes de caractères, et les classes `template` basic_fstream et basic_stringstream, dérivées de la classe basic_iostream, pour les flux d'entrée / sortie sur les fichiers et les chaînes de caractères.

> **Note :** Cette hiérarchie de classes est assez complexe et peut paraître étrange au niveau des classes des flux mixtes. En effet, la classe basic_fstream ne dérive pas des classes basic_ifstream et basic_ofstream mais de la classe basic_iostream, et c'est cette classe qui dérive des classes basic_istream et basic_ostream. De même, la classe basic_stringstream ne dérive pas des classes basic_istringstream et basic_ostringstream, mais de la classe basic_iostream.

Comme il l'a déjà été dit, toutes ces classes `template` peuvent être instanciées pour n'importe quel type de caractère, pourvu qu'une classe de traits char_traits soit définie. Cependant, en pratique, il n'est courant d'instancier ces classes que pour les types de caractères de base du langage, à savoir les types char et wchar_t.

Historiquement, les classes d'entrée / sortie des bibliothèques fournies avec la plupart des implémentations n'étaient pas `template` et ne permettaient de manipuler que des flux basés sur le type de caractère char. Les implémentations disposant de classes de flux d'entrée / sortie capables de manipuler les caractères de type wchar_t étaient donc relativement rares. À présent, toutes ces classes sont définies comme des instances des classes `template` citées ci-dessus. Par souci de compatibilité, la bibliothèque standard C++ définit tout un jeu de types pour ces instances qui permettent aux programmes utilisant les anciennes classes de fonctionner. Ces types sont déclarés de la manière suivante dans l'en-tête `iosfwd` (mais sont définis dans leurs en-têtes respectifs, que l'on décrira plus tard) :

```
// Types de base des tampons :
typedef basic_streambuf<char>    streambuf;
typedef basic_streambuf<wchar_t> wstreambuf;
typedef basic_stringbuf<char>    stringbuf;
typedef basic_stringbuf<wchar_t> wstringbuf;
typedef basic_filebuf<char>      filebuf;
typedef basic_filebuf<wchar_t>   wfilebuf;

// Types de base des flux d'entrée / sortie :
typedef basic_ios<char>    ios;
typedef basic_ios<wchar_t> wios;

// Types des flux d'entrée / sortie standards :
typedef basic_istream<char>     istream;
typedef basic_istream<wchar_t>  wistream;
typedef basic_ostream<char>     ostream;
typedef basic_ostream<wchar_t>  wostream;
typedef basic_iostream<char>    iostream;
typedef basic_iostream<wchar_t> wiostream;

// Types des flux orientés fichiers :
typedef basic_ifstream<char>    ifstream;
typedef basic_ifstream<wchar_t> wifstream;
typedef basic_ofstream<char>    ofstream;
```

```
typedef basic_ofstream<wchar_t> wofstream;
typedef basic_fstream<char>     fstream;
typedef basic_fstream<wchar_t>  wfstream;

// Types des flux orientés chaînes de caractères :
typedef basic_istringstream<char>    istringstream;
typedef basic_istringstream<wchar_t> wistringstream;
typedef basic_ostringstream<char>    ostringstream;
typedef basic_ostringstream<wchar_t> wostringstream;
typedef basic_stringstream<char>     stringstream;
typedef basic_stringstream<wchar_t>  wstringstream;
```

Les objets `cin`, `cout`, `cerr` et `clog` sont donc des instances des classes istream et ostream, qui sont associées aux flux d'entrée / sortie standards du programme. En fait, la bibliothèque standard définit également des versions capables de manipuler des flux basés sur le type wchar_t pour les programmes qui désirent travailler avec des caractères larges. Ces objets sont respectivement `wcin` (instance de wistream), `wcout`, `wcerr` et `wclog` (instances de wostream). Tous ces objets sont initialisés par la bibliothèque standard automatiquement lorsqu'ils sont utilisés pour la première fois, et sont donc toujours utilisables.

> **Note :** En réalité, sur la plupart des systèmes, les flux d'entrée / sortie standards sont les premiers descripteurs de fichiers que le système attribue automatiquement aux programmes lorsqu'ils sont lancés. En toute logique, les objets `cin`, `cout`, `cerr` et `clog` devraient donc être des instances de classes de gestion de flux orientés fichiers. Cependant, ces fichiers ne sont pas nommés d'une part et, d'autre part, tous les systèmes ne gèrent pas les flux d'entrée / sortie standards de la même manière. Ces objets ne sont donc pas toujours des flux sur des fichiers et la bibliothèque standard C++ ne les définit par conséquent pas comme tels.

15.2. Les tampons

Les classes de gestion des tampons de la bibliothèque standard C++ se situent au cœur des opérations d'écriture et de lecture sur les flux de données physiques qu'un programme est susceptible de manipuler. Bien qu'elles ne soient quasiment jamais utilisées directement par les programmeurs, c'est sur ces classes que les classes de flux s'appuient pour effectuer les opérations d'entrée sortie. Il est donc nécessaire de connaître un peu leur mode de fonctionnement.

15.2.1. Généralités sur les tampons

Un *tampon*, également appelé *cache*, est une zone mémoire dans laquelle les opérations d'écriture et de lecture se font et dont le contenu est mis en correspondance avec les données d'un média physique sous-jacent. Les mécanismes de cache ont essentiellement pour but d'optimiser les performances des opérations d'entrée / sortie. En effet, l'accès à la mémoire cache est généralement beaucoup plus rapide que l'accès direct au support physique ou au média de communication. Les opérations effectuées par le programme se font donc, la plupart du temps, uniquement au niveau du tampon, et ce n'est que dans certaines conditions que les données du tampon sont effectivement transmises au média physique. Le gain en performance peut intervenir à plusieurs niveau. Les cas les plus simples étant simplement lorsqu'une donnée écrite est écrasée peu de temps après par une autre valeur (la première opération d'écriture n'est alors jamais transmise au média) ou lorsqu'une donnée est lue

plusieurs fois (la même donnée est renvoyée à chaque lecture). Bien entendu, cela suppose que les données stockées dans le tampon soient cohérentes avec les données du média, surtout si les données sont accédées au travers de plusieurs tampons. Tout mécanisme de gestion de cache permet donc de *vider* les caches (c'est-à-dire de forcer les opérations d'écriture) et de les *invalider* (c'est-à-dire de leur signaler que leurs données sont obsolètes et qu'une lecture physique doit être faite si on cherche à y accéder).

Les mécanismes de mémoire cache et de tampon sont très souvent utilisés en informatique, à tous les niveaux. On trouve des mémoires cache dans les processeurs, les contrôleurs de disque, les graveurs de CD, les pilotes de périphériques des systèmes d'exploitation et bien entendu dans les programmes. Chacun de ces caches contribue à l'amélioration des performances globales en retardant au maximum la réalisation des opérations lentes et en optimisant les opérations de lecture et d'écriture (souvent en les effectuant en groupe, ce qui permet de réduire les frais de communication ou d'initialisation des périphériques). Il n'est donc absolument pas surprenant que la bibliothèque standard C++ utilise elle aussi la notion de tampon dans toutes ses classes d'entrée / sortie...

15.2.2. La classe basic_streambuf

Les mécanismes de base des tampons de la bibliothèque standard sont implémentés dans la classe `template` basic_streambuf. Cette classe n'est pas destinée à être utilisée telle quelle car elle ne sait pas communiquer avec les supports physiques des données. En fait, elle ne peut être utilisée qu'en tant que classe de base de classes plus spécialisées, qui elles fournissent les fonctionnalités d'accès aux médias par l'intermédiaire de fonctions virtuelles. La classe `basic_streambuf` appelle donc ces méthodes en diverses circonstances au sein des traitements effectués par son propre code de gestion du tampon, aussi bien pour signaler les changements d'état de celui-ci que pour demander l'écriture ou la lecture des données dans la séquence sous contrôle.

La classe basic_streambuf fournit donc une interface publique permettant d'accéder aux données du tampon d'un côté et définit l'interface de communication avec ses classes filles par l'intermédiaire de ses méthodes virtuelles de l'autre coté. Bien entendu, ces méthodes virtuelles sont toutes déclarées en zone protégée afin d'éviter que l'on puisse les appeler directement, tout en permettant aux classes dérivées de les redéfinir et d'y accéder.

En interne, la classe basic_streambuf encapsule deux tampons, un pour les écritures et un pour les lectures. Cependant, ces tampons accèdent à la même mémoire et à la même séquence de données physiques. Ces deux tampons peuvent être utilisés simultanément ou non, suivant la nature de la séquence sous contrôle et suivant le flux qui utilise le tampon. Par exemple, les flux de sortie n'utilisent que le tampon en écriture, et les flux d'entrée que le tampon en lecture.

La classe basic_streambuf gère ses tampons d'entrée et de sortie à l'aide d'une zone de mémoire interne qui contient un sous-ensemble des données de la séquence sous contrôle. Les deux tampons travaillent de manière indépendante sur cette zone de mémoire et sont chacun représentés à l'aide de trois pointeurs. Ces pointeurs contiennent respectivement l'adresse du début de la zone mémoire du tampon, son adresse de fin et l'adresse de la position courante en lecture ou en écriture. Ces pointeurs sont complètement gérés en interne par la classe basic_streambuf, mais les classes dérivées peuvent y accéder et les modifier en fonction de leurs besoins par l'intermédiaire d'accesseurs. Les pointeurs d'un tampon peuvent parfaitement être nuls si celui-ci n'est pas utilisé. Toutefois, si le pointeur référençant la position courante n'est pas nul, ses pointeurs associés ne doivent pas l'être et la position courante référencée doit obligatoirement se situer dans une zone mémoire définie par les pointeurs de début et de fin du tampon.

La classe basic_streambuf est déclarée comme suit dans l'en-tête `streambuf` :

```
template <class charT, class traits =
```

```
    char_traits<charT> >
class basic_streambuf
{
public:
// Les types de base :
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;

    // Les méthodes publiques utilisables par les classes de flux :

// Les méthodes de gestion des locales :
    locale pubimbue(const locale &loc);
    locale getloc() const;

// Les méthodes de gestion du tampon :
    basic_streambuf<char_type,traits> *
        pubsetbuf(char_type* s, streamsize n);
    pos_type pubseekoff(off_type off, ios_base::seekdir sens,
        ios_base::openmode mode = ios_base::in | ios_base::out);
    pos_type pubseekpos(pos_type sp,
        ios_base::openmode mode = ios_base::in | ios_base::out);
    int pubsync();

// Méthodes d'accès au tampon en lecture :
    streamsize in_avail();
    int_type sgetc();
    int_type sbumpc();
    int_type snextc();
    streamsize sgetn(char_type *s, streamsize n);

// Méthode d'annulation de lecture d'un caractère :
    int_type sputbackc(char_type c);
    int_type sungetc();

// Méthode d'accès en écriture :
    int_type   sputc(char_type c);
    streamsize sputn(const char_type *s, streamsize n);

    // Le destructeur :
    virtual ~basic_streambuf();

protected:
    // Les méthodes protected utilisables par
    // les classes dérivées :

// Le constructeur :
    basic_streambuf();

// Méthodes d'accès aux pointeurs du tampon de lecture :
    char_type *eback() const;
    char_type *gptr()  const;
    char_type *egptr() const;
    void      gbump(int n);
    void      setg(char_type *debut, char_type *suivant,
```

```
        char_type *fin);

// Méthodes d'accès aux pointeurs du tampon d'écriture :
    char_type *pbase() const;
    char_type *pptr() const;
    char_type *epptr() const;
    void      pbump(int n);
    void      setp(char_type *debut, char_type *fin);

    // Les méthodes protected virtuelles, que les classes
    // dérivées doivent implémenter :

    virtual void imbue(const locale &loc);
    virtual basic_streambuf<char_type, traits>*
        setbuf(char_type *s, streamsize n);
    virtual pos_type seekoff(off_type off, ios_base::seekdir sens,
        ios_base::openmode mode = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp,
        ios_base::openmode mode = ios_base::in | ios_base::out);
    virtual int        sync();
    virtual int        showmanyc();
    virtual streamsize xsgetn(char_type *s, streamsize n);
    virtual int_type   underflow();
    virtual int_type   uflow();
    virtual int_type   pbackfail(int_type c = traits::eof());
    virtual streamsize xsputn(const char_type* s, streamsize n);
    virtual int_type   overflow (int_type c = traits::eof());
};
```

Comme vous pouvez le constater, le constructeur de la classe basic_streambuf est déclaré en zone `protected`, ce qui empêche quiconque de l'instancier. C'est normal, puisque cette classe n'est destinée à être utilisée qu'en tant que classe de base d'une classe spécialisée pour un média spécifique. En revanche, les méthodes virtuelles ne sont pas pures, car elles fournissent un comportement par défaut qui conviendra dans la plupart des cas.

L'interface publique comprend des méthodes d'ordre générale et des méthodes permettant d'effectuer les opérations d'écriture et de lecture sur les tampons encapsulés par la classe basic_streambuf. Pour les distinguer des méthodes virtuelles qui doivent être implémentées dans les classes dérivées, leur nom est préfixé par `pub` (pour « publique »).

Les méthodes `pubimbue` et `locale` permettent respectivement de fixer la locale utilisée par le programme pour ce tampon et de récupérer la locale courante. Par défaut, la locale globale active au moment de la construction du tampon est utilisée. Les notions de locales seront décrites dans le Chapitre 16.

Les méthodes `pubsetbuf`, `pubseekoff` et `pubseekpos` permettent quant à elles de paramétrer le tampon d'entrée / sortie. Ces méthodes se contentent d'appeler les méthodes virtuelles `setbuf`, `seekoff` et `seekpos`, dont le comportement, spécifique à chaque classe dérivée, sera décrit ci-dessous.

Viennent ensuite les méthodes d'accès aux données en lecture et en écriture. Les méthodes de lecture sont respectivement les méthodes `sgetc`, `sbumpc` et `snextc`. La méthode `sgetc` permet de lire la valeur du caractère référencé par le pointeur courant du tampon d'entrée. Cette fonction renvoie la même valeur à chaque appel, car elle ne modifie pas la valeur de ce pointeur. En revanche, la méthode `sbumpc` fait avancer ce pointeur après la lecture du caractère courant, ce qui fait qu'elle peut être utilisée pour lire tous les caractères du flux de données physiques. Enfin, la méthode `snextc` appelle

la méthode `sbumpc` dans un premier temps, puis renvoie la valeur retournée par `sgetc`. Cette méthode permet donc de lire la valeur du caractère suivant dans le tampon et de positionner le pointeur sur ce caractère. Notez que, contrairement à la méthode `sbumpc`, `snextc` modifie la valeur du pointeur avant de lire le caractère courant, ce qui fait qu'en réalité elle lit le caractère suivant. Toutes ces méthodes renvoient la valeur de fin de fichier définie dans la classe des traits du type de caractère utilisé en cas d'erreur. Elles sont également susceptibles de demander la lecture de données complémentaires dans le cadre de la gestion du tampon.

La méthode `in_avail` renvoie le nombre de caractères encore stockés dans le tampon géré par la classe basic_streambuf. Si ce tampon est vide, elle renvoie une estimation du nombre de caractères qui peuvent être lus dans la séquence contrôlée par le tampon. Cette estimation est un minimum, la valeur renvoyée garantit qu'autant d'appel à `sbumpc` réussiront.

Les tampons de la bibliothèque standard donnent la possibilité aux programmes qui les utilisent d'annuler la lecture d'un caractère. Normalement, ce genre d'annulation ne peut être effectué qu'une seule fois et la valeur qui doit être replacée dans le tampon doit être exactement celle qui avait été lue. Les méthodes qui permettent d'effectuer ce type d'opération sont les méthodes `sputbackc` et `sungetc`. La première méthode prend en paramètre la valeur du caractère qui doit être replacé dans le tampon et la deuxième ne fait que décrémenter le pointeur référençant l'élément courant dans le tampon de lecture. Ces deux opérations peuvent échouer si la valeur à replacer n'est pas égale à la valeur du caractère qui se trouve dans le tampon ou si, tout simplement, il est impossible de revenir en arrière (par exemple parce qu'on se trouve en début de séquence). Dans ce cas, ces méthodes renvoient la valeur de fin de fichier définie dans la classe des traits du type de caractère utilisé, à savoir `traits::eof()`.

Enfin, les méthodes d'écriture de la classe basic_streambuf sont les méthodes `sputc` et `sputn`. La première permet d'écrire un caractère unique dans le tampon de la séquence de sortie et la deuxième d'écrire toute une série de caractères. Dans ce dernier cas, les caractères à écrire sont spécifiés à l'aide d'un tableau dont la longueur est passée en deuxième paramètre à `sputn`. Ces deux méthodes peuvent renvoyer le caractère de fin de fichier de la classe des traits du type de caractère utilisé pour signaler une erreur d'écriture.

L'interface protégée de la classe basic_streambuf est constituée des accesseurs aux pointeurs sur les tampons d'entrée et de sortie d'une part et, d'autre part, des méthodes virtuelles que les classes dérivées doivent redéfinir pour implémenter la gestion des entrées / sorties physiques.

Les pointeurs du tableau contenant les données du tampon de lecture peuvent être récupérés par les classes dérivées à l'aide des méthodes `eback`, `gptr` et `egptr`. La méthode `eback` renvoie le pointeur sur le début du tableau du tampon d'entrée. Les méthodes `gptr` et `egptr` renvoient quant à elles le pointeur sur l'élément courant, dont la valeur peut être obtenue avec la méthode `sgetc`, et le pointeur sur la fin du tableau du tampon. Le nom de la méthode `gptr` provient de l'abréviation de l'anglais « get pointer » et celui de la méthode `egptr` de l'abréviation « end of gptr ». Enfin, les méthodes `gbump` et `setg` permettent respectivement de faire avancer le pointeur sur l'élément courant d'un certain nombre de positions et de fixer les trois pointeurs du tampon de lecture en une seule opération.

Les pointeurs du tampon d'écriture sont accessibles par des méthodes similaires à celles définies pour le tampon de lecture. Ainsi, les méthodes `pbase`, `pptr` et `epptr` permettent respectivement d'accéder au début du tableau contenant les données du tampon d'écriture, à la position courante pour les écritures et au pointeur de fin de ce tableau. « pptr » est ici l'abréviation de l'anglais « put pointer ». Les méthodes `pbump` et `setp` jouent le même rôle pour les pointeurs du tampon d'écriture que les méthodes `gbump` et `setg` pour les pointeurs du tampon de lecture.

Enfin, les méthodes protégées de la classe basic_streambuf permettent, comme on l'a déjà indiqué ci-dessus, de communiquer avec les classes dérivées implémentant les entrées / sorties physiques. Le rôle de ces méthodes est décrit dans le tableau ci-dessous :

Méthode	Description
`imbue`	Cette méthode est appelée à chaque fois qu'il y a un changement de locale au niveau du tampon. Les classes dérivées de la classe basic_streambuf sont assurées qu'il n'y aura pas de changement de locale entre chaque appel à cette fonction et peuvent donc maintenir une référence sur la locale courante en permanence. Les notions concernant les locales seront décrites dans le Chapitre 16.
`setbuf`	Cette méthode n'est appelée que par la méthode `pubsetbuf`. Elle a principalement pour but de fixer la zone mémoire utilisée par le tampon. Ceci peut ne pas avoir de sens pour certains médias, aussi cette méthode peut-elle ne rien faire du tout. En pratique, si cette fonction est appellée avec deux paramètres nuls, les mécanismes de gestion du cache doivent être désactivés. Pour cela, la classe dérivée doit fixer les pointeurs des tampons de lecture et d'écriture à la valeur nulle.
`seekoff`	Cette méthode n'est appelée que par la méthode `pubseekoff`. Tout comme la méthode `setbuf`, sa sémantique est spécifique à chaque classe dérivée de gestion des médias physiques. En général, cette fonction permet de déplacer la position courante dans la séquence de données d'un certain décalage. Ce décalage peut être spécifié relativement à la position courante, au début ou à la fin de la séquence de données sous contrôle. Le mode de déplacement est spécifié à l'aide du paramètre `sens`, qui doit prendre l'une des constantes de type seekdir définie dans la classe ios_base. De même, le tampon concerné par ce déplacement est spécifié par le paramètre `mode`, dont la valeur doit être l'une des constante de type ios_base::openmode. Ces types et ces constantes seront décrits avec la classe de base ios_base dans la Section 15.3.
`seekpos`	Cette méthode n'est appelée que par la méthode `pubseekpos`. Elle fonctionne de manière similaire à la méthode `seekoff` puisqu'elle permet de positionner le pointeur courant des tampons de la classe basic_streambuf à un emplacement arbitraire dans la séquence de données sous contrôle.
`sync`	Cette méthode n'est appelée que par la méthode `pubsync` et permet de demander la synchronisation du tampon avec la séquence de données physiques. Autrement dit, les opérations d'écritures doivent être effectuées sur le champ afin de s'assurer que les modifiations effectuées dans le cache soient bien enregistrées.
`showmanyc`	Cette méthode est appelée par la méthode `in_avail` lorsque la fin du tampon de lecture a été atteinte. Elle doit renvoyer une estimation basse du nombre de caractères qui peuvent encore être lus dans la séquence sous contrôle. Cette estimation doit être sûre, dans le sens où le nombre de caractères renvoyés doit effectivement pouvoir être lu sans erreur.
`xsgetn`	Cette méthode n'est appelée que par la méthode `sgetn`. Elle permet d'effectuer la lecture de plusieurs caractères et de les stocker dans le tableau reçu en paramètre. La lecture de chaque caractère doit se faire exactement comme si la méthode `sbumpc` était appelée successivement pour chacun d'eux afin de maintenir le tampon de lecture dans un état cohérent. La valeur retournée est le nombre de caractères effectivement lus ou `traits::eof()` en cas d'erreur.

Méthode	Description
`underflow`	Cette méthode est appelée lorsque la fin du tampon est atteinte lors de la lecture d'un caractère. Cela peut se produire lorsqu'il n'y a plus de caractère disponible dans le tampon ou tout simplement à chaque lecture, lorsque le mécanisme de cache est désactivé. Cette fonction doit renvoyer le caractère suivant de la séquence sous contrôle et remplir le tampon si nécessaire. Le pointeur référençant le caractère courant est alors initialisé sur le caractère dont la valeur a été récupérée. Ainsi, la méthode `underflow` doit remplir le tampon, mais ne doit pas faire avancer la position courante de lecture. Cette méthode peut renvoyer `traits::eof()` en cas d'échec, ce qui se produit généralement lorsque la fin de la séquence sous contrôle a été atteinte.
`uflow`	Cette méthode est appelée dans les mêmes conditions que la méthode `underflow`. Elle doit également remplir le tampon, mais, contrairement à `underflow`, elle fait également avancer le pointeur du caractère courant d'une position. Ceci implique que cette méthode ne peut pas être utilisée avec les flux non bufferisés. En général, cette méthode n'a pas à être redéfinie parce que le code de la méthode `uflow` de la classe basic_streambuf effectue ces opérations en s'appuyant sur la méthode `underflow`. Cette méthode peut renvoyer `traits::eof()` en cas d'échec.
`pbackfail`	Cette méthode est appelée lorsque la méthode `sputbackc` échoue, soit parce que le pointeur de lecture se trouve au début du tampon de lecture, soit parce que le caractère qui doit être replacé dans la séquence n'est pas le caractère qui vient d'en être extrait. Cette méthode doit prendre en charge le déplacement des caractères du tampon pour permettre le replacement du caractère fourni en paramètre et mettre à jour les pointeurs de gestion du tampon de lecture en conséquence. Elle peut renvoyer la valeur `traits::eof()` pour signaler un échec.
`xsputn`	Cette méthode n'est appelée que par la méthode `sputn`. Elle permet de réaliser l'écriture de plusieurs caractères dans la séquence de sortie. Ces caractères sont spécifiés dans le tableau fourni en paramètre. Ces écritures sont réalisées exactement comme si la méthode `sputc` était appelée successivement pour chaque caractère afin de maintenir le tampon d'écriture dans un état cohérent. La valeur retournée est le nombre de caractères écrits ou `traits::eof()` en cas d'erreur.
`overflow`	Cette méthode est appelée par les méthodes d'écriture de la classe basic_streambuf lorsque le tampon d'écriture est plein. Elle a pour but de dégager de la place dans ce tampon en consommant une partie des caractères situés entre le pointeur de début du tampon et le pointeur de position d'écriture courante. Elle est donc susceptible d'effectuer les écritures physiques sur le média de sortie au cours de cette opération. Si l'écriture réussit, les pointeurs de gestion du tampon d'écriture doivent être mis à jour en conséquence. Dans le cas contraire, la fonction peut renvoyer la valeur `traits::eof()` pour signaler l'erreur ou lancer une exception.

15.2.3. Les classes de tampons basic_stringbuf et

basic_filebuf

Vous l'aurez compris, l'écriture d'une classe dérivée de la classe basic_streambuf prenant en charge un média peut être relativement technique et difficile. Heureusement, cette situation ne se présente quasiment jamais, parce que la bibliothèque standard C++ fournit des classes dérivées prenant en charge les deux situations les plus importantes : les tampons d'accès à une chaîne de caractères et les tampons d'accès aux fichiers. Ces classes sont respectivement les classes `template` basic_stringbuf et basic_filebuf.

15.2.3.1. La classe basic_stringbuf

La classe basic_stringbuf permet d'effectuer des entrées / sorties en mémoire de la même manière que si elles étaient effectuées sur un périphérique d'entrée / sortie normal. Le but de cette classe n'est évidemment pas d'optimiser les performances à l'aide d'un cache puisque les opérations se font à destination de la mémoire, mais d'uniformiser et de permettre les mêmes opérations de formatage dans des chaînes de caractères que celles que l'on peut réaliser avec les flux d'entrée / sortie normaux.

La classe basic_streambuf dérive bien entendu de la classe basic_streambuf puisqu'elle définit les opérations fondamentales d'écriture et de lecture dans une chaîne de caractères. Elle est déclarée comme suit dans l'en-tête `sstream` :

```
template <class charT,
    class traits = char_traits<charT>,
    class Allocator = allocator<chatT> >
class basic_stringbuf : public basic_streambuf<charT, traits>
{
public:
// Les types :
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;

// Les constructeurs / destructeurs :
    explicit basic_stringbuf(
        ios_base::openmode mode = ios_base::in | ios_base::out);
    explicit basic_stringbuf(
        const basic_string<charT,traits,Allocator> &str,
        ios_base::openmode mode = ios_base::in | ios_base::out);
    virtual ~basic_stringbuf();

// Les méthodes de gestion de la chaîne de caractères sous contrôle :
    basic_string<charT,traits,Allocator> str() const;
    void str(const basic_string<charT,traits,Allocator> &s);
};
```

Comme cette déclaration le montre, la classe basic_streambuf définit elle aussi un jeu de types permettant d'obtenir facilement les types de objets manipulés. De plus, elle définit également quelques méthodes complémentaires permettant d'effectuer les opérations spécifiques aux flux orientés chaîne de caractères. En particulier, les constructeurs permettent de fournir une chaîne de caractères à partir de laquelle le tampon sera initialisé. Cette chaîne de caractères est copiée lors de la construction du tampon, ce qui fait qu'elle peut être réutilisée ou modifiée après la création du tampon. Ces constructeurs prennent également en paramètre le mode de fonctionnement du tampon. Ce mode peut être la lecture

(auquel cas le paramètre `mode` vaut `ios_base::in`), l'écriture (`mode` vaut alors `ios_base::out`) ou les deux (`mode` vaut alors la combinaison de ces deux constantes par un ou logique). Les constantes de mode d'ouverture sont définis dans la classe ios_base, que l'on décrira dans la Section 15.3.

> **Note :** Vous remarquerez que, contrairement au constructeur de la classe basic_streambuf, les constructeurs de la classe basic_stringbuf sont déclarés dans la zone de déclaration publique, ce qui autorise la création de tampons de type basic_stringbuf. Le constructeur de la classe de base est appelé par ces constructeurs, qui ont le droit de le faire puisqu'il s'agit d'une méthode `protected`.

Il est également possible d'accéder aux données stockées dans le tampon à l'aide des accesseurs `str`. Le premier renvoie une basic_string contenant la chaîne du tampon en écriture si possible (c'est-à-dire si le tampon a été créé dans le mode écriture ou lecture / écriture), et la chaîne du tampon en lecture sinon. Le deuxième accesseur permet de définir les données du tampon a posteriori, une fois celui-ci créé.

Exemple 15-1. Lecture et écriture dans un tampon de chaîne de caractères

```
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

int main(void)
{
    // Construit une chaîne de caractère :
    string s("123456789");
    // Construit un tampon basé sur cette chaîne :
    stringbuf sb(s);
    // Lit quelques caractères unitairement :
    cout << (char) sb.sbumpc() << endl;
    cout << (char) sb.sbumpc() << endl;
    cout << (char) sb.sbumpc() << endl;
    // Replace le dernier caractère lu dans le tampon :
    sb.sungetc();
    // Lit trois caractères consécutivement :
    char tab[4];
    sb.sgetn(tab, 3);
    tab[3] = 0;
    cout << tab << endl;
    // Écrase le premier caractère de la chaîne :
    sb.sputc('a');
    // Récupère une copie de la chaîne utilisée par le tampon :
    cout << sb.str() << endl;
    return 0;
}
```

> **Note :** La classe basic_stringbuf redéfinit bien entendu certaines des méthodes protégées de la classe basic_streambuf. Ces méthodes n'ont pas été présentées dans la déclaration ci-dessus parce qu'elles font partie de l'implémentation de la classe basic_stringbuf et leur description n'a que peu d'intérêt pour les utilisateurs.

15.2.3.2. La classe basic_filebuf

La classe basic_filebuf est la classe qui prend en charge les opérations d'entrée / sortie sur fichier dans la bibliothèque standard C++.

Pour la bibliothèque standard C++, un fichier est une séquence de caractères simples (donc de type char). Il est important de bien comprendre qu'il n'est pas possible, avec la classe basic_filebuf, de manipuler des fichiers contenant des données de type wchar_t. En effet, même dans le cas où les données enregistrées sont de type wchar_t, les fichiers contenant ces données sont enregistrés sous la forme de séquences de caractères dont l'unité de base reste le caractère simple. La manière de coder les caractères larges dans les fichiers n'est pas spécifiée et chaque implémentation est libre de faire ce qu'elle veut à ce niveau. Généralement, l'encodage utilisé est un encodage à taille variable, c'est à dire que chaque caractère large est représenté sous la forme d'un ou de plusieurs caractères simples, selon sa valeur et selon sa position dans le flux de données du fichier.

Cela signifie qu'il ne faut pas faire d'hypothèse sur la manière dont les instances de la classe `template` basic_filebuf enregistrent les données des fichiers pour des valeurs du paramètre `template charT` autres que le type char. En général, l'encodage utilisé ne concerne pas le programmeur, puisqu'il suffit d'enregistrer et de lire les fichiers avec les même types de classes basic_filebuf pour retrouver les données initiales. Toutefois, si les fichiers doivent être relus par des programmes écrits dans un autre langage ou compilés avec un autre compilateur, il peut être nécessaire de connaître l'encodage utilisé. Vous trouverez cette information dans la documentation de votre environnement de développement.

La classe basic_filebuf est déclarée comme suit dans l'en-tête `fstream` :

```
template <class charT,
    class traits = char_traits<charT> >
class basic_filebuf : public basic_streambuf<charT,traits>
{
public:
// Les types :
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;

// Les constructeurs / destructeurs :
    basic_filebuf();
    virtual ~basic_filebuf();

// Les méthodes de gestion du fichier sous contrôle :
    basic_filebuf<charT,traits> *open(const char *s,
        ios_base::openmode mode);
    basic_filebuf<charT,traits> *close();
    bool is_open() const;
};
```

Comme vous pouvez le constater, la classe basic_filebuf est semblable à la classe basic_stringbuf. Outre les déclarations de types et celles du constructeur et du destructeur, elle définit trois méthodes permettant de réaliser les opérations spécifiques aux fichiers.

La méthode `open` permet, comme son nom l'indique, d'ouvrir un fichier. Cette méthode prend en paramètre le nom du fichier à ouvrir ainsi que le mode d'ouverture. Ce mode peut être une combinaison logique de plusieurs constantes définies dans la classe ios_base. Ces constantes sont décrites dans la

Section 15.3. Les plus importantes sont `in`, qui permet d'ouvrir un fichier en lecture, `out`, qui permet de l'ouvrir en lecture, `binary`, qui permet de l'ouvrir en mode binaire, `app`, qui permet de l'ouvrir en mode ajout, et `trunc`, qui permet de le vider lorsqu'il est ouvert en écriture. La méthode `open` renvoie le pointeur `this` si le fichier a pu être ouvert ou le pointeur nul dans le cas contraire.

La classe basic_filebuf ne gère qu'une seule position pour la lecture et l'écriture dans les fichiers. Autrement dit, si un fichier est ouvert à la fois en lecture et en écriture, les pointeurs de lecture et d'écriture du tampon auront toujours la même valeur. L'écriture à une position provoquera donc non seulement la modification de la position courante en écriture, mais également celle de la position courante en lecture.

La méthode `close` est la méthode à utiliser pour fermer un fichier ouvert. Cette méthode ne peut fonctionner que si un fichier est effectivement ouvert dans ce tampon. Elle renvoie le pointeur `this` si le fichier courant a effectivement pu être fermé ou le pointeur nul en cas d'erreur.

Enfin, la méthode `is_open` permet de déterminer si un fichier est ouvert ou non dans ce tampon.

Exemple 15-2. Lecture et écriture dans un tampon de fichier

```
#include <iostream>
#include <string>
#include <fstream>

using namespace std;

int main(void)
{
    // Ouvre un fichier texte et crée un tampon pour y accéder :
    filebuf fb;
    fb.open("test.txt", ios_base::in | ios_base::out | ios_base::trunc);
    // Teste si le fichier est ouvert :
    if (fb.is_open())
    {
        // Écrit deux lignes :
        string l1 = "Bonjour\n";
        string l2 = "tout le monde !\n";
        fb.sputn(l1.data(), l1.size());
        fb.sputn(l2.data(), l2.size());
        // Repositionne le pointeur de fichier au début.
        // Note : le déplacement se fait pour les deux
        // tampons parce qu'il n'y a qu'un pointeur
        // sur les données du fichier :
        fb.pubseekpos(0, ios_base::in | ios_base::out);
        // Lit les premières lettres du fichier :
        cout << (char) fb.sbumpc() << endl;
        cout << (char) fb.sbumpc() << endl;
        cout << (char) fb.sbumpc() << endl;
        // Ferme le fichier :
        fb.close();
    }
    return 0;
}
```

15.3. Les classes de base des flux : ios_base et basic_ios

Les classes de gestion des flux constituent la deuxième hiérarchie de classes de la bibliothèque standard d'entrée / sortie. Bien que destinées à accéder à des médias variés, ces classes disposent d'une interface commune qui permet d'en simplifier l'utilisation. Cette interface est essentiellement définie par deux classes de bases fondamentales : la classe ios_base, qui définit toutes les fonctionnalités indépendantes du type de caractère utilisé par les flux, et la classe `template` basic_ios, qui regroupe l'essentiel des fonctionnalités des flux d'entrée / sortie.

15.3.1. La classe ios_base

La classe ios_base est une classe C++ classique dont toutes les classes `template` de gestion des flux d'entrée / sortie dérivent. Cette classe ne fournit, comme c'est le cas de la plupart des classes de base, qu'un nombre de fonctionnalités très réduit. En pratique, sa principale utilité est de définir plusieurs jeux de constantes qui sont utilisées par ses classes dérivées pour identifier les options des différents modes de fonctionnement disponibles. Ces constantes portent un nom standardisé mais leur type n'est pas précisé par la norme C++. Cependant, leur nature (entière, énumération, champ de bits) est imposée, et les implémentations doivent définir un `typedef` permettant de créer des variables du même type.

La classe de base ios_base est déclarée comme suit dans l'en-tête `ios` :

```
class ios_base
{
// Constructeur et destructeur :
protected:
    ios_base();
public:
    ~ios_base();

// Classe de base des exceptions des flux d'entrée / sortie :
    class failure;

// Classe d'initialisation des objets d'entrée / sortie standards :
    class Init;

// Constantes de définition des options de formatage :
    typedef T1 fmtflags;
    static const fmtflags boolalpha;
    static const fmtflags hex;
    static const fmtflags oct;
    static const fmtflags dec;
    static const fmtflags fixed;
    static const fmtflags scientific;
    static const fmtflags left;
    static const fmtflags right;
    static const fmtflags internal;
    static const fmtflags showbase;
    static const fmtflags showpoint;
    static const fmtflags showpos;
    static const fmtflags uppercase;
    static const fmtflags unitbuf;
    static const fmtflags skipws;
```

```
    static const fmtflags adjustfield;
    static const fmtflags basefield;
    static const fmtflags floatfield;

// Constantes des modes d'ouverture des flux et des fichiers :
    typedef T3 openmode;
    static const openmode in;
    static const openmode out;
    static const openmode binary;
    static const openmode trunc;
    static const openmode app;
    static const openmode ate;

// Constantes de définition des modes de positionnement :
    typedef T4 seekdir;
    static const seekdir beg;
    static const seekdir cur;
    static const seekdir end;

// Constantes d'état des flux d'entrée / sortie :
    typedef T2 iostate;
    static const iostate goodbit;
    static const iostate eofbit;
    static const iostate failbit;
    static const iostate badbit;

// Accesseurs sur les options de formatage :
    fmtflags flags() const;
    fmtflags flags(fmtflags fmtfl);
    fmtflags setf(fmtflags fmtfl);
    fmtflags setf(fmtflags fmtfl, fmtflags mask);
    void unsetf(fmtflags mask);
    streamsize precision() const;
    streamsize precision(streamsize prec);
    streamsize width() const;
    streamsize width(streamsize wide);

// Méthode de synchronisation :
    static bool sync_with_stdio(bool sync = true);

// Méthode d'enregistrement des callback pour les événements :
    enum event { erase_event, imbue_event, copyfmt_event };
    typedef void (*event_callback)(event, ios_base &, int index);
    void register_callback(event_call_back fn, int index);

// Méthode de gestion des données privées :
    static int xalloc();
    long &iword(int index);
    void* &pword(int index);

// Méthodes de gestion des locales :
    locale imbue(const locale &loc);
    locale getloc() const;
};
```

Comme vous pouvez le constater, le constructeur de la classe ios_base est déclaré en zone protégée. Il n'est donc pas possible d'instancier un objet de cette classe, ce qui est normal puisqu'elle n'est destinée qu'à être la classe de base de classes plus spécialisées.

Le premier jeu de constantes défini par la classe ios_base contient toutes les valeurs de type fmtflags, qui permettent de spécifier les différentes options à utiliser pour le formatage des données écrites dans les flux. Ce type doit obligatoirement être un champ de bits. Les constantes quant à elles permettent de définir la base de numérotation utilisée, si celle-ci doit être indiquée avec chaque nombre ou non, ainsi que les différentes options de formatage à utiliser. La signification précise de chacune de ces constantes est donnée dans le tableau suivant :

Tableau 15-1. Options de formatage des flux

Constante	Signification
`boolalpha`	Permet de réaliser les entrées / sorties des booléens sous forme textuelle et non sous forme numérique. Ainsi, les valeurs `true` et `false` ne sont pas écrites ou lues sous la forme de `0` ou de `1`, mais sous la forme fixée par la classe de localisation utilisée par le flux. Par défaut, les booléens sont représentés par les chaînes de caractères « true » et « false » lorsque ce flag est actif. Cependant, il est possible de modifier ces chaînes de caractères en définissant une locale spécifique. Les notions de locales seront décrites dans le Chapitre 16.
`hex`	Permet de réaliser les entrées / sorties des entiers en base hexadécimale.
`oct`	Permet de réaliser les entrées / sorties des entiers en base octale.
`dec`	Permet de réaliser les entrées / sorties des entiers en décimal.
`fixed`	Active la représentation en virgule fixe des nombres à virgule flottante.
`scientific`	Active la représentation en virgule flottante des nombres à virgule flottante.
`left`	Utilise l'alignement à gauche pour les données écrites sur les flux de sortie. Dans le cas où la largeur des champs est fixée, des caractères de remplissage sont ajoutés à la droite de ces données pour atteindre cette largeur.
`right`	Utilise l'alignement à droite pour les données écrites sur les flux de sortie. Dans le cas où la largeur des champs est fixée, des caractères de remplissage sont ajoutés à la gauche de ces données pour atteindre cette largeur.
`internal`	Effectue un remplissage avec les caractères de remplissage à une position fixe déterminée par la locale en cours d'utilisation si la largeur des données est inférieure à la largeur des champs à utiliser. Si la position de remplissage n'est pas spécifiée par la locale pour l'opération en cours, le comportement adopté est l'alignement à droite.
`showbase`	Précise la base utilisée pour le formatage des nombres entiers.
`showpoint`	Écrit systématiquement le séparateur de la virgule dans le formatage des nombres à virgule flottante, que la partie fractionnaire de ces nombres soit nulle ou non. Le caractère utilisé pour représenter ce séparateur est défini dans la locale utilisée par le flux d'entrée / sortie. La notion de locale sera vue dans le Chapitre 16. Par défaut, le caractère utilisé est le point décimal (' . ').
`showpos`	Utilise systématiquement le signe des nombres écrits sur le flux de sortie, qu'ils soient positifs ou négatifs. Le formatage du signe des nombre se fait selon les critères définis par la locale active pour ce flux. Par défaut, les nombres négatifs sont préfixés du symbole '−' et les nombres positifs du symbole '+' si cette option est active. Dans le cas contraire, le signe des nombres positifs n'est pas écrit.

Constante	Signification
`uppercase`	Permet d'écrire en majuscule certains caractères, comme le '`e`' de l'exposant des nombres à virgule flottante par exemple, ou les chiffres hexadécimaux `A` à `F`.
`unitbuf`	Permet d'effectuer automatiquement une opération de synchronisation du cache utilisé par le flux de sortie après chaque écriture.
`skipws`	Permet d'ignorer les blancs précédant les données à lire dans les opérations d'entrée pour lesquelles de tels blancs sont significatifs.

La classe ios_base définit également les constantes `adjustfield`, `basefield` et `floatfield`, qui sont en réalité des combinaisons des autres constantes. Ainsi, la constante `adjustfield` représente l'ensemble des options d'alignement (à savoir `left`, `right` et `internal`), la constante `basefield` représente les options de spécification de base pour les sorties numériques (c'est-à-dire les options `hex`, `oct` et `dec`), et la constante `floatfield` les options définissant les types de formatage des nombres à virgules (`scientific` et `fixed`).

Le deuxième jeu de constantes permet de caractériser les modes d'ouverture des flux et des fichiers. Le type de ces constantes est le type openmode. Il s'agit également d'un champ de bits, ce qui permet de réaliser des combinaisons entre leurs valeurs pour cumuler différents modes d'ouverture lors de l'utilisation des fichiers. Les constantes définies par la classe ios_base sont décrites dans le tableau ci-dessous :

Tableau 15-2. Modes d'ouverture des fichiers

Constante	Signification
`in`	Permet d'ouvrir le flux en écriture.
`out`	Permet d'ouvrir le flux en lecture.
`binary`	Permet d'ouvrir le flux en mode binaire, pour les systèmes qui font une distinction entre les fichiers textes et les fichiers binaires. Ce flag n'est pas nécessaire pour les systèmes d'exploitation conformes à la norme POSIX. Cependant, il est préférable de l'utiliser lors de l'ouverture de fichiers binaires si l'on veut que le programme soit portable sur les autres systèmes d'exploitation.
`trunc`	Permet de vider automatiquement le fichier lorsqu'une ouverture en écriture est demandée.
`app`	Permet d'ouvrir le fichier en mode ajout lorsqu'une ouverture en écriture est demandée. Dans ce mode, le pointeur de fichier est systématiquement positionné en fin de fichier avant chaque écriture. Ainsi, les écritures se font les unes à la suite des autres, toujours à la fin du fichier, et quelles que soient les opérations qui peuvent avoir lieu sur le fichier entre-temps.
`ate`	Permet d'ouvrir le fichier en écriture et de positionner le pointeur de fichier à la fin de celui-ci. Notez que ce mode de fonctionnement se distingue du mode `app` par le fait que si un repositionnement a lieu entre deux écritures la deuxième écriture ne se fera pas forcément à la fin du fichier.

Le troisième jeu de constantes définit les diverses directions qu'il est possible d'utiliser lors d'un repositionnement d'un pointeur de fichier. Le type de ces constantes, à savoir le type seekdir, est une énumération dont les valeurs sont décrites dans le tableau ci-dessous :

Tableau 15-3. Directions de déplacement dans un fichier

Constante	Signification
`beg`	Le déplacement de fait par rapport au début du fichier. Le décalage spécifié dans les fonctions de repositionnement doit être positif ou nul, la valeur 0 correspondant au début du fichier.
`cur`	Le déplacement se fait relativement à la position courante. Le décalage spécifié dans les fonctions de repositionnement peut donc être négatif, positif ou nul (auquel cas aucun déplacement n'est effectué).
`end`	Le déplacement se fait relativement à la fin du fichier. Le décalage fourni dans les fonctions de repositionnement doit être positif ou nul, la valeur 0 correspondant à la fin de fichier.

Enfin, les constantes de type iostate permettent de décrire les différents états dans lequel un flux d'entrée / sortie peut se trouver. Il s'agit, encore une fois, d'un champ de bits, et plusieurs combinaisons sont possibles.

Tableau 15-4. États des flux d'entrée / sortie

Constante	Signification
`goodbit`	Cette constante correspond à l'état normal du flux, lorsqu'il ne s'est produit aucune erreur.
`eofbit`	Ce bit est positionné dans la variable d'état du flux lorsque la fin du flux a été atteinte, soit parce qu'il n'y a plus de données à lire, soit parce qu'on ne peut plus en écrire.
`failbit`	Ce bit est positionné dans la variable d'état du flux lorsqu'une erreur logique s'est produite lors d'une opération de lecture ou d'écriture. Ceci peut avoir lieu lorsque les données écrites ou lues sont incorrectes.
`badbit`	Ce bit est positionné lorsqu'une erreur fatale s'est produite. Ce genre de situation peut se produire lorsqu'une erreur a eu lieu au niveau matériel (secteur défectueux d'un disque dur ou coupure réseau par exemple).

Les différentes variables d'état des flux d'entrée / sortie peuvent être manipulées à l'aide de ces constantes et des accesseurs de la classe ios_base. Les méthodes les plus importantes sont sans doute celles qui permettent de modifier les options de formatage pour le flux d'entrée / sortie. La méthode `flags` permet de récupérer la valeur de la variable d'état contenant les options de formatage du flux. Cette méthode dispose également d'une surcharge qui permet de spécifier une nouvelle valeur pour cette variable d'état, et qui retourne la valeur précédente. Il est aussi possible de fixer et de désactiver les options de formatage indépendamment les unes des autres à l'aide des méthodes `setf` et `unsetf`. La méthode `setf` prend en paramètre les nouvelles options qui doivent être ajoutées au jeu d'options déjà actives, avec, éventuellement, un masque permettant de réinitialiser certaines autres options. On emploiera généralement un masque lorsque l'on voudra fixer un paramètre parmi plusieurs paramètres mutuellement exclusifs, comme la base de numérotation utilisée par exemple. La méthode `unsetf` prend quant à elle le masque des options qui doivent être supprimées du jeu d'options utilisé par le flux en paramètre.

Outre les méthodes de gestion des options de formatage, la classe ios_base définit deux surcharges pour chacune des méthodes `precision` et `width`. Ces méthodes permettent respectivement de lire et de fixer la précision avec laquelle les nombres à virgule doivent être écrits et la largeur minimale

des conversions des nombres lors des écritures.

La plupart des options que l'on peut fixer sont permanentes, c'est-à-dire qu'elles restent actives jusqu'à ce qu'on spécifie de nouvelles options. Cependant, ce n'est pas le cas du paramètre de largeur que l'on renseigne grâce à la méthode `width`. En effet, chaque opération d'écriture réinitialise ce paramètre à la valeur 0. Il faut donc spécifier la largeur minimale pour chaque donnée écrite sur le flux.

Exemple 15-3. Modification des options de formatage des flux

```
#include <iostream>
using namespace std;

// Affiche un booléen, un nombre entier et un nombre à virgule :
void print(bool b, int i, float f)
{
    cout << b << " " << i << " " << f << endl;
}

int main(void)
{
    // Affiche avec les options par défaut :
    print(true, 35, 3105367.9751447);
    // Passe en hexadécimal :
    cout.unsetf(ios_base::dec);
    cout.setf(ios_base::hex);
    print(true, 35, 3105367.9751447);
    // Affiche la base des nombres et
    // affiche les booléens textuellement :
    cout.setf(ios_base::boolalpha);
    cout.setf(ios_base::showbase);
    print(true, 35, 3105367.9751447);
    // Affiche un flottant en notation à virgule fixe
    // avec une largeur minimale de 16 caractères :
    cout << "***";
    cout.width(16);
    cout.setf(ios_base::fixed, ios_base::floatfield);
    cout << 315367.9751447;
    cout << "***" << endl;
    // Recommence en fixant la précision
    // à 3 chiffres et la largeur à 10 caractères :
    cout << "***";
    cout.precision(3);
    cout.width(10);
    cout << 315367.9751447;
    cout << "***" << endl;
    return 0;
}
```

Note : On prendra bien garde au fait que la largeur des champs dans lesquels les données sont écrites est une largeur minimale, pas une largeur maximale. En particulier, cela signigie que les écritures ne sont pas tronquées si elles sont plus grande que cette largeur. On devra donc faire extrêmement attention à ne pas provoquer de débordements lors des écritures.

On n'oubliera pas de s'assurer de la cohérence des paramètres du flux lorsqu'on modifie la valeur d'une option. Par exemple, dans l'exemple précédent, il faut désactiver l'emploi de la numérotation décimale lorsque l'on demande à utiliser la base hexadécimale. Cette opération a été faite

explicitement ici pour bien montrer son importance, mais elle aurait également pu être réalisée par l'emploi d'un masque avec la constante `ios_base::basefield`. L'exemple précédent montre comment utiliser un masque avec l'appel à `setf` pour fixer la représentation des nombres à virgule.

La classe ios_base fournit également un certain nombre de services généraux au programmeur et à ses classes dérivées. La méthode `sync_with_stdio` permet de déterminer, pour un flux d'entrée / sortie standard, s'il est synchronisé avec le flux sous-jacent ou si des données se trouvent encore dans son tampon. Lorsqu'elle est appelée avec le paramètre `false` dès le début du programme, elle permet de décorréler le fonctionnement du flux C++ et du flux standard sous-jacent. Pour tous les autres appels, cette méthode ignore le paramètre qui lui est fourni. La méthode `register_callback` permet d'enregistrer une fonction de rappel qui sera appelée par la classe ios_base lorsque des événements susceptibles de modifier notablement le comportement du flux se produisent. Ces fonctions de rappel peuvent recevoir une valeur entière en paramètre qui peut être utilisée pour référencer des données privées contenant des paramètres plus complexes. Les méthodes `xalloc`, `iword` et `pword` sont fournies afin de permettre de stocker ces données privées et de les retrouver facilement à l'aide d'un indice, qui peut être la valeur passée en paramètre à la fonction de rappel. Ces méthodes permettent de récupérer des références sur des valeurs de type long et sur des pointeurs de type void. Enfin, la classe ios_base fournit une classe de base pour les exceptions que les classes de flux pourront utiliser afin de signaler une erreur et une classe permettant d'initialiser les objets `cin`, `cout`, `cerr`, `clog` et leurs semblables pour les caractères larges. Toutes ces fonctionnalités ne sont généralement pas d'une très grande utilité pour les programmeurs et sont en réalité fournie pour faciliter l'implémentation des classes de flux de la bibliothèque standard.

Enfin, la classe ios_base fournit les méthodes `getloc` et `imbue` qui permettent respectivement de récupérer la locale utilisée par le flux et d'en fixer une autre. Cette locale est utilisée par le flux pour déterminer la manière de représenter et de lire les nombres et pour effectuer les entrées / sorties formatées en fonction des paramètres de langue et des conventions locales du pays où le programme est exécuté. Les notions de locale et de paramètres internationaux seront décrits en détail dans le Chapitre 16.

15.3.2. La classe basic_ios

La classe `template` basic_ios fournit toutes les fonctionnalités communes à toutes les classes de flux de la bibliothèque d'entrée / sortie. Cette classe dérive de la classe ios_base et apporte tous les mécanismes de gestion des tampons pour les classes de flux. La classe basic_ios est déclarée comme suit dans l'en-tête `ios` :

```
template <class charT,
    class traits = char_traits<charT> >
class basic_ios : public ios_base
{
// Constructeur et destructeur :
protected:
    basic_ios();
    void init(basic_streambuf<charT,traits> *flux);

public:
// Types de données :
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
```

```
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;

// Constructeur publique, destructeur et opération de copie :
    explicit basic_ios(basic_streambuf<charT,traits> *flux);
    virtual ~basic_ios();
    basic_ios &copyfmt(const basic_ios &);

// Méthodes de gestion des tampons :
    basic_streambuf<charT,traits> *rdbuf() const;
    basic_streambuf<charT,traits> *rdbuf(
        basic_streambuf<charT,traits> *tampon);

// Méthodes de gestion des exceptions :
    iostate exceptions() const;
    void exceptions(iostate except);

// Accesseurs :
    operator void*() const
    bool operator!() const
    iostate rdstate() const;
    void clear(iostate statut = goodbit);
    void setstate(iostate statut);
    bool good() const;
    bool eof()  const;
    bool fail() const;
    bool bad()  const;
    char_type fill() const;
    char_type fill(char_type c);
    basic_ostream<charT,traits> *tie() const;
    basic_ostream<charT,traits> *tie(
        basic_ostream<charT,traits> *flux);

// Méthodes de gestion des locales :
    locale imbue(const locale &loc);
    char   narrow(char_type c, char defaut) const;
    char_type widen(char c) const;
};
```

Le constructeur de base ainsi que la méthode `init`, destinée à associer un tampon à un flux après sa construction, sont déclarés en zone protégée. Ainsi, il n'est pas possible d'instancier et d'initialiser manuellement un flux avec un tampon. Cependant, la classe basic_ios fournit un constructeur en zone publique qui permet de créer un nouveau flux et de l'initialiser à la volée. Ce constructeur prend en paramètre l'adresse d'un tampon qui sera associé au flux après la construction. Remarquez que, bien qu'il soit ainsi parfaitement possible d'instancier un objet de type l'une des instances de la classe `template` basic_ios, cela n'a pas grand intérêt. En effet, cette classe ne fournit pas assez de fonctionnalités pour réaliser des entrées / sorties facilement sur le flux. La classe basic_ios est donc réellement destinée à être utilisée en tant que classe de base pour des classes plus spécialisées dans les opérations d'entrée / sortie.

Une fois initialisés, les flux sont associés aux tampons grâce auxquels ils accèdent aux médias physiques pour leurs opérations d'entrée / sortie. Cependant, cette association n'est pas figée et il est possible de changer de tampon a posteriori. Les manipulations de tampon sont effectuées avec les deux surcharges de la méthode `rdbuf`. La première permet de récupérer l'adresse de l'objet tampon

courant et la deuxième d'en spécifier un nouveau. Cette dernière méthode renvoie l'adresse du tampon précédent. Bien entendu, le fait de changer le tampon d'un flux provoque sa réinitialisation.

Les flux de la bibliothèque standard peuvent signaler les cas d'erreurs aux fonctions qui les utilisent de différentes manières. La première est simplement de renvoyer un code d'erreur (`false` ou le caractère de fin de fichier), et la deuxième est de lancer une exception dérivée de la classe d'exception failure (définie dans la classe de base ios_base). Ce comportement est paramétrable en fonction des types d'erreurs qui peuvent se produire. Par défaut, les classes de flux n'utilisent pas les exceptions, quelles que soient les erreurs rencontrées. Toutefois, il est possible d'activer le mécanisme des exceptions individuellement pour chaque type d'erreur possible. La classe basic_ios gère pour cela un masque d'exceptions qui peut être récupéré et modifié à l'aide de deux méthodes surchargées. Ces méthodes sont les méthodes `exceptions`. La première version renvoie le masque courant et la deuxième permet de fixer un nouveau masque. Les masques d'exceptions sont constitués de combinaisons logiques des bits d'état des flux définis dans la classe ios_base (à savoir `goodbit`, `eofbit`, `failbit` et `badbit`). Le fait de changer le masque d'exceptions réinitialise l'état du flux.

La classe basic_ios fournit également tout un ensemble d'accesseurs grâce auxquels il est possible de récupérer l'état courant du flux. Ces accesseurs sont principalement destinés à faciliter la manipulation du flux et à simplifier les différentes expressions dans lesquelles il est utilisé. Par exemple, l'opérateur de transtypage vers le type pointeur sur void permet de tester la validité du flux comme s'il s'agissait d'un pointeur. Cet opérateur retourne en effet une valeur non nulle si le flux est utilisable (c'est-à-dire si la méthode `fail` renvoie `false`. De même, l'opérateur de négation `operator!` renvoie la même valeur que la méthode `fail`.

Comme vous l'aurez sans doute compris, la méthode `fail` indique si le flux (et donc le tampon contrôlé par ce flux) est dans un état correct. En pratique, cette méthode renvoie `true` dès que l'un des bits `ios_base::failbit` ou `ios_base::badbit` est positionné dans la variable d'état du flux. Vous pourrez faire la distinction entre ces deux bits grâce à la méthode `bad`, qui elle ne renvoie `true` que si le bit `ios_base::badbit` est positionné. Les autres méthodes de lecture de l'état du flux portent des noms explicites et leur signification ne doit pas poser de problème. On prendra toutefois garde à bien distinguer la méthode `clear`, qui permet de réinitialiser l'état du flux avec le masque de bits passé en paramètre, de la méthode `setstate`, qui permet de positionner un bit complémentaire. Ces deux méthodes sont susceptibles de lancer des exceptions si le nouvel état du flux le requiert et si son masque d'exceptions l'exige.

Le dernier accesseur utile pour le programmeur est l'accesseur `fill`. Cet accesseur permet de lire la valeur du caractère de remplissage utilisé lorsque la largeur des champs est supérieure à la largeur des données qui doivent être écrites sur le flux de sortie. Par défaut, ce caractère est le caractère d'espacement.

> **Note :** Les deux surcharges de la méthode `tie` permettent de stocker dans le flux un pointeur sur un flux de sortie standard avec lequel les opérations d'entrée / sortie doivent être synchronisées. Ces méthodes sont utilisées en interne par les méthodes d'entrée / sortie des classes dérivées de la classe basic_ios et ne sont pas réellement utiles pour les programmeurs. En général donc, seule les classes de la bibliothèque standard les appelleront.

Enfin, la classe basic_ios prend également en compte la locale du flux dans tous ses traitements. Elle redéfinit donc la méthode `imbue` afin de pouvoir détecter les changement de locale que l'utilisateur peut faire. Bien entendu, la méthode `getloc` est héritée de la classe de base ios_base et permet toujours de récupérer la locale courante. De plus, la classe basic_ios définit deux méthodes permettant de réaliser les conversions entre le type de caractère char et le type de caractère fourni en paramètre `template`. La méthode `widen` permet, comme son nom l'indique, de convertir un caractère de type `char` en un caractère du type `template` du flux. Inversement, la méthode `narrow` permet de conver-

tir un caractère du type de caractère du flux en un caractère de type char. Cette méthode prend en paramètre le caractère à convertir et la valeur par défaut que doit prendre le résultat en cas d'échec de la conversion.

15.4. Les flux d'entrée / sortie

La plupart des fonctionnalités des flux d'entrée / sortie sont implémentées au niveau des classes `template` basic_ostream et basic_istream. Ces classes dérivent toutes deux directement de la classe basic_ios, dont elles héritent de toutes les fonctionnalités de gestion des tampons et de gestion d'état.

Les classes basic_ostream et basic_istream seront sans doute les classes de flux que vous utiliserez le plus souvent, car c'est à leur niveau que sont définies toutes les fonctionnalités de lecture et d'écriture sur les flux, aussi bien pour les données formatées telles que les entiers, les flottants ou les chaînes de caractères, que pour les écritures de données brutes. De plus, les flux d'entrée / sortie standards `cin`, `cout`, `cerr` et `clog` sont tous des instances de ces classes.

La bibliothèque standard définit également une classe capable de réaliser à la fois les opérations de lecture et d'écriture sur les flux : la classe basic_iostream. En fait, cette classe dérive simplement des deux classes basic_istream et basic_ostream, et regroupe donc toutes les fonctionnalités de ces deux classes.

15.4.1. La classe de base basic_ostream

La classe basic_ostream fournit toutes les fonctions permettant d'effectuer des écritures sur un flux de sortie, que ces écritures soient formatées ou non. Elle est déclarée comme suit dans l'en-tête `ostream` :

```
template <class charT,
    class traits = char_traits<charT> >
class basic_ostream : virtual public basic_ios<charT, traits>
{
public:
// Les types de données :
    typedef charT                    char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                   traits_type;

// Le constructeur et le destructeur :
    explicit basic_ostream(basic_streambuf<char_type, traits> *tampon);
    virtual ~basic_ostream();

// Les opérations d'écritures formatées :
    basic_ostream<charT, traits> &operator<<(bool);
    basic_ostream<charT, traits> &operator<<(short);
    basic_ostream<charT, traits> &operator<<(unsigned short);
    basic_ostream<charT, traits> &operator<<(int);
    basic_ostream<charT, traits> &operator<<(unsigned int);
    basic_ostream<charT, traits> &operator<<(long);
    basic_ostream<charT, traits> &operator<<(unsigned long);
    basic_ostream<charT, traits> &operator<<(float);
    basic_ostream<charT, traits> &operator<<(double);
```

```
    basic_ostream<charT, traits> &operator<<(long double);
    basic_ostream<charT, traits> &operator<<(void *);
    basic_ostream<charT, traits> &operator<<
          (basic_streambuf<char_type, traits> *tampon);

// Classe de gestion des exceptions pour les opérateurs d'écritures formatées :
    class sentry
    {
    public:
        explicit sentry(basic_ostream<charT, traits> &);
        ~sentry();
        operator bool();
    };

// Les opérations d'écritures non formatées :
    basic_ostream<charT, traits> &put(char_type);
    basic_ostream<charT, traits> &write(const char_type *p, streamsize taille);

// Les opérations de gestion du tampon :
    basic_ostream<charT, traits> &flush();
    pos_type tellp();
    basic_ostream<charT, traits> &seekp(pos_type);
    basic_ostream<charT, traits> &seekp(off_type, ios_base::seekdir);

// Les opérations de gestion des manipulateurs :
    basic_ostream<charT, traits> &operator<<
          (basic_ostream<charT, traits> & (*pf)(
       basic_ostream<charT, traits> &));
    basic_ostream<charT, traits> &operator<<
          (basic_ios<charT, traits> & (*pf)(basic_ios<charT, traits> &));
    basic_ostream<charT, traits> &operator<<
          (ios_base & (*pf)(ios_base &));
};
```

Comme vous pouvez le constater, le constructeur de cette classe prend en paramètre un pointeur sur l'objet tampon dans lequel les écritures devront être réalisées. Vous pouvez donc construire un flux de sortie à partir de n'importe quel tampon, simplement en fournissant ce tampon en paramètre au constructeur. Cependant, il ne faut pas procéder ainsi en général, mais utiliser plutôt les classes dérivées de la classe basic_ostream et spécialisées dans les écritures sur fichiers et dans des chaînes de caractères.

Les écritures formatées sont réalisées par l'intermédiaire de différentes surcharges de l'opérateur d'insertion `operator<<`, dont il existe une version pour chaque type de donnée de base du langage. Ainsi, l'écriture d'une valeur dans le flux de sortie se fait extrêmement simplement :

```
// Écriture d'une chaîne de caractères sur le flux de sortie standard :
cout << "Voici la valeur d'un entier :\n";

// Écriture d'un entier sur le flux de sortie standard :
cout << 45;
```

Vous constaterez que, grâce aux mécanismes de surcharge, ces écritures se font exactement de la même manière pour tous les types de données. On ne peut faire plus simple...

> **Note :** Les opérations de formatage prennent en compte toutes les options de formatage qui sont stockées dans la classe de base ios_base. En général, les opérations d'écriture ne modifient pas ces options. Toutefois, la largeur minimale des champs dans lesquels les résultats sont formatés est systématiquement réinitialisée à 0 après chaque écriture. Il est donc nécessaire, lorsque l'on réalise plusieurs écritures formatées dans un flux de sortie, de spécifier pour chaque valeur sa largeur minimale si elle ne doit pas être égale à 0. Autrement dit, il faut redire la largeur de chaque champ, même s'ils utilisent tous la même largeur minimale.
>
> Les opérations de formatage utilisent également les conventions locales du pays où le programme est exécuté, conventions qui sont définies dans la locale incluse dans le flux de sortie. Les notions de locale seront détaillées dans le Chapitre 16.

Bien entendu, il est possible de définir de nouvelles surcharges de l'opérateur d'insertion pour les types définis par l'utilisateur, ce qui permet d'étendre à l'infini les possibilités de cette classe. Ces surcharges devront obligatoirement être définies à l'extérieur de la classe `template` basic_ostream et, si l'on veut les écrire de manière générique, elles devront également être des fonctions `template` paramétrées par le type de caractère du flux sur lequel elles travaillent.

Vous noterez la présence d'une classe sentry dans la classe basic_ostream. Cette classe est une classe utilitaire permettant de réaliser les initialisations qui doivent précéder toutes les opérations d'écriture au sein des surcharges de l'opérateur d'insertion. Entre autres opérations, le constructeur de cette classe peut synchroniser le flux de sortie standard encapsulé dans le flux grâce à la méthode `tie` de la classe de base basic_ios, et prendre toutes les mesures devant précéder les écritures sur le flux. Vous devrez donc toujours utiliser un objet local de ce type lorsque vous écrirez une surcharge de l'opérateur `operator<<` pour vos propres types. Les écritures ne devront être réalisées que si l'initialisation a réussi, ce qui peut être vérifié simplement en comparant l'objet local de type sentry avec la valeur `true`.

Exemple 15-4. Définition d'un nouvel opérateur d'insertion pour un flux de sortie

```
#include <iostream>
#include <string>

using namespace std;

// Définition d'un type de donnée privé :
struct Personne
{
    string Nom;
    string Prenom;
    int Age;       // En centimètres.
    int Taille;
};

// Définition de l'opérateur d'écriture pour ce type :
template <class charT, class Traits>
basic_ostream<charT, Traits> &operator<<(
    basic_ostream<charT, Traits> &flux,
    const Personne &p)
{
    // Inialisation du flux de sortie :
    typename basic_ostream<charT, Traits>::sentry init(flux);
    if (init)
    {
        // Écriture des données :
```

```
        int Metres = p.Taille / 100;
        int Reste = p.Taille % 100;
        flux << p.Prenom << " " << p.Nom <<
            " mesure " << Metres <<
            "m" << Reste << " (" <<
            p.Age << " an";
        if (p.Age > 1) flux << "s";
        flux << ")";
    }
    return flux;
}

int main(void)
{
    // Construit une nouvelle personne :
    Personne p;
    p.Nom = "Dupont";
    p.Prenom = "Jean";
    p.Age = 28;
    p.Taille = 185;
    // Affiche les caractéristiques de cette personne :
    cout << p << endl;
    return 0;
}
```

Note : L'utilisation de l'objet local de type sentry comme un booléen est autorisée parce que la classe sentry définit un opérateur de transtypage vers le type bool.

Le constructeur de la classe sentry est susceptible de lancer des exceptions, selon la configuration du masque d'exceptions du flux de sortie avec lequel on l'initialise.

Les écritures de données brutes ne disposent bien entendu pas de surcharges pour chaque type de donnée, puisqu'il s'agit dans ce cas d'écrire les données directement sur le flux de sortie, sans les formater sous forme textuelle. Ces écritures sont donc réalisées par l'intermédiaire de méthodes dédiées qui effectuent soit l'écriture d'un caractère unique, soit l'écriture d'un tableau de caractères complet. Pour écrire un unique caractère sur le flux de sortie, vous pouvez utiliser la méthode `put`. Pour l'écriture d'un bloc de données en revanche, il faut utiliser la méthode `write`, qui prend en paramètre un pointeur sur la zone de données à écrire et la taille de cette zone.

Exemple 15-5. Écriture de données brutes sur un flux de sortie

```
#include <iostream>
using namespace std;

// Définition de quelques codes de couleurs
// pour les terminaux ANSI :
const char Rouge[] = {033, '[', '3', '1', 'm'};
const char Vert[] = {033, '[', '3', '2', 'm'};
const char Jaune[] = {033, '[', '3', '3', 'm'};
const char Reset[] = {033, '[', 'm', 017};

int main(void)
{
    // Écriture d'un message coloré :
    cout.write(Rouge, sizeof(Rouge));
```

```
    cout << "Bonjour ";
    cout.write(Vert, sizeof(Vert));
    cout << "tout ";
    cout.write(Jaune, sizeof(Jaune));
    cout << "le monde !" << endl;
    cout.write(Reset, sizeof(Reset));
    return 0;
}
```

Bien entendu, la classe basic_ostream fournit les méthodes nécessaires à la gestion du tampon sous-jacent. La méthode `flush` permet de synchroniser le tampon utilisé par le flux (en appelant la méthode `pubsync` de ce dernier). La méthode `tellp` permet de lire la position courante dans le flux de sortie, et les deux surcharges de la méthode `seekp` permettent de modifier cette position soit de manière absolue, soit de manière relative à la position courante. Nous verrons un exemple d'utilisation de ces méthodes dans la description des classes de flux pour les fichiers.

La classe basic_ostream définit également des surcharges de l'opérateur d'insertion capables de prendre en paramètre des pointeurs de fonctions. Ces méthodes ne constituent pas des opérations d'écriture à proprement parler, mais permettent de réaliser des opérations sur les flux de sortie plus facilement à l'aide de fonctions capables de les manipuler. En raison de cette propriété, ces fonctions sont couramment appelées des *manipulateurs*. En réalité, ces manipulateurs ne sont rien d'autre que des fonctions prenant un flux en paramètre et réalisant des opérations sur ce flux. Les opérateurs `operator<<` prenant en paramètre ces manipulateurs les exécutent sur l'objet courant `*this`. Ainsi, il est possible d'appliquer ces manipulateurs à un flux simplement en réalisant une écriture du manipulateur sur ce flux, exactement comme pour les écritures normales.

La bibliothèque standard définit tout un jeu de manipulateurs extrêmement utiles pour définir les options de formatage et pour effectuer des opérations de base sur les flux. Grâce à ces manipulateurs, il n'est plus nécessaire d'utiliser la méthode `setf` de la classe ios_base par exemple. Par exemple, le symbole `endl` utilisé pour effectuer un retour à la ligne dans les opérations d'écriture sur les flux de sortie n'est rien d'autre qu'un manipulateur, dont la déclaration est la suivante :

```
template <class charT, class Traits>
basic_ostream<charT, Traits> &endl(
    basic_ostream<charT, Traits> &flux);
```

et dont le rôle est simplement d'écrire le caractère de retour à la ligne et d'appeler la méthode `flush` du flux de sortie.

Il existe des manipulateurs permettant de travailler sur la classe de base ios_base ou sur ses classes dérivées comme la classe basic_ostream par exemple, d'où la présence de plusieurs surcharges de l'opérateur d'insertion pour ces différents manipulateurs. Il existe également des manipulateurs prenant des paramètres et renvoyant un type de données spécial pour lequel un opérateur d'écriture a été défini, et qui permettent de réaliser des opérations plus complexes nécessitant des paramètres complémentaires. Les manipulateurs sont définis, selon leur nature, soit dans l'en-tête de déclaration du flux, soit dans l'en-tête `ios`, soit dans l'en-tête `iomanip`.

Le tableau suivant présente les manipulateurs les plus simples qui ne prennent pas de paramètre :

Tableau 15-5. Manipulateurs des flux de sortie

Manipulateur	Fonction
`endl`	Envoie un caractère de retour à la ligne sur le flux et synchronise le tampon par un appel à la méthode `flush`.
`ends`	Envoie un caractère nul terminal de fin de ligne sur le flux.

Manipulateur	Fonction
`flush`	Synchronise le tampon utilisé par le flux par un appelle à la méthode `flush`.
`boolalpha`	Active le formatage des booléens sous forme textuelle.
`noboolalpha`	Désactive le formatage textuel des booléens.
`hex`	Formate les nombres en base 16.
`oct`	Formate les nombres en base 8.
`dec`	Formate les nombres en base 10.
`fixed`	Utilise la notation en virgule fixe pour les nombres à virgule.
`scientific`	Utilise la notation en virgule flottante pour les nombres à virgule.
`left`	Aligne les résultats à gauche.
`right`	Aligne les résultats à droite.
`internal`	Utilise le remplissage des champs avec des espaces complémentaires à une position fixe déterminée par la locale courante. Équivalent à `right` si la locale ne spécifie aucune position de remplissage particulière.
`showbase`	Indique la base de numérotation utilisée.
`noshowbase`	N'indique pas la base de numérotation utilisée.
`showpoint`	Utilise le séparateur de virgule dans les nombres à virgule, même si la partie fractionnaire est nulle.
`noshowpoint`	N'utilise le séparateur de virgule que si la partie fractionnaire des nombres à virgule flottante est significative.
`showpos`	Écrit systématiquement le signe des nombres, même s'ils sont positifs.
`noshowpos`	N'écrit le signe des nombres que s'ils sont négatifs.
`uppercase`	Écrit les exposants et les chiffres hexadécimaux en majuscule.
`nouppercase`	Écrit les exposants et les chiffres hexadécimaux en minuscule.
`unitbuf`	Effectue une opération de synchronisation du cache géré par le tampon du flux après chaque écriture.
`nounitbuf`	N'effectue les opérations de synchronisation du cache géré par le tampon du flux que lorsque cela est explicitement demandé.

Les paramètres suivants sont un peu plus complexes, puisqu'ils prennent des paramètres complémentaires. Ils renvoient un type de donnée spécifique à chaque implémentation de la bibliothèque standard et qui n'est destiné qu'à être inséré dans un flux de sortie à l'aide de l'opérateur d'insertion :

Tableau 15-6. Manipulateurs utilisant des paramètres

Manipulateur	Fonction
`resetiosflags(ios_base::fmtflags)`	Permet d'effacer certains bits des options du flux. Ces bits sont spécifiés par une combinaison logique de constantes de type ios_base::fmtflags.
`setiosflags(ios_base::fmtflags)`	Permet de positionner certains bits des options du flux. Ces bits sont spécifiés par une combinaison logique de constantes de type ios_base::fmtflags.

Manipulateur	Fonction
`setbase(int base)`	Permet de sélectionner la base de numérotation utilisée. Les valeurs admissibles sont 8, 10 et 16 respectivement pour la base octale, la base décimale et la base hexadécimale.
`setprecision(int)`	Permet de spécifier la précision (nombre de caractères significatifs) des nombres formatés.
`setw(int)`	Permet de spécifier la largeur minimale du champ dans lequel la donnée suivante sera écrite à la prochaine opération d'écriture sur le flux.
`setfill(char_type)`	Permet de spécifier le caractère de remplissage à utiliser lorsque la largeur des champs est inférieure à la largeur minimale spécifiée dans les options de formatage.

Exemple 15-6. Utilisation des manipulateurs sur un flux de sortie

```
#include <iostream>
#include <iomanip>

using namespace std;

int main(void)
{
    // Affiche les booléens sous forme textuelle :
    cout << boolalpha << true << endl;
    // Écrit les nombres en hexadécimal :
    cout << hex << 57 << endl;
    // Repasse en base 10 :
    cout << dec << 57 << endl;
    // Affiche un flottant avec une largeur
    // minimale de 15 caractères :
    cout << setfill('*') << setw(15) << 3.151592 << endl;
    // Recommence mais avec un alignement à gauche :
    cout << left << setw(15) << 3.151592 << endl;
}
```

15.4.2. La classe de base basic_istream

La deuxième classe la plus utilisée de la bibliothèque d'entrée / sortie est sans doute la classe `template` basic_istream. À l'instar de la classe ostream, cette classe fournit toutes les fonctionnalités de lecture de données formatées ou non à partir d'un tampon. Ce sont donc certainement les méthodes cette classe que vous utiliserez le plus souvent lorsque vous désirerez lire les données d'un flux. La classe basic_istream est déclarée comme suit dans l'en-tête `istream` :

```
template <class charT,
    class traits = char_traits<charT> >
class basic_istream : virtual public basic_ios<charT, traits>
{
public:
```

```
// Les types de données :
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;

// Le constructeur et destructeur :
    explicit basic_istream(basic_streambuf<charT, traits> *sb);
    virtual ~basic_istream();

// Les opération de gestion des entrées formatées :
    basic_istream<charT, traits> &operator>>(bool &n);
    basic_istream<charT, traits> &operator>>(short &n);
    basic_istream<charT, traits> &operator>>(unsigned short &n);
    basic_istream<charT, traits> &operator>>(int &n);
    basic_istream<charT, traits> &operator>>(unsigned int &n);
    basic_istream<charT, traits> &operator>>(long &n);
    basic_istream<charT, traits> &operator>>(unsigned long &n);
    basic_istream<charT, traits> &operator>>(float &f);
    basic_istream<charT, traits> &operator>>(double &f);
    basic_istream<charT, traits> &operator>>(long double &f);
    basic_istream<charT, traits> &operator>>(void * &p);
    basic_istream<charT, traits> &operator>>
        (basic_streambuf<char_type, traits> *sb);

// Classe de gestion des exceptions pour les opérateurs d'écritures formatées :
    class sentry
    {
    public:
        explicit sentry(basic_istream<charT, traits> &flux,
            bool conserve = false);
        ~sentry();
        operator bool();
    };

// Les opérations de lecture des données brutes :
    int_type get();
    basic_istream<charT, traits> &get(char_type &c);
    int_type peek();
    basic_istream<charT, traits> &putback(char_type c);
    basic_istream<charT, traits> &unget();
    basic_istream<charT, traits> &read(char_type *s, streamsize n);
    streamsize                   readsome(char_type *s, streamsize n);

    basic_istream<charT, traits> &get(char_type *s, streamsize n);
    basic_istream<charT, traits> &get(char_type *s, streamsize n,
        char_type delim);
    basic_istream<charT, traits> &get(
        basic_streambuf<char_type, traits> &sb);
    basic_istream<charT, traits> &get(
        basic_streambuf<char_type, traits> &sb, char_type delim);
    basic_istream<charT, traits> &getline(char_type *s, streamsize n);
    basic_istream<charT, traits> &getline(char_type *s, streamsize n,
        char_type delim);
    basic_istream<charT, traits> &ignore
        (streamsize n = 1, int_type delim = traits::eof());
```

```
    streamsize gcount() const;

// Les opérations de gestion du tampon :
    int sync();
    pos_type tellg();
    basic_istream<charT, traits> &seekg(pos_type);
    basic_istream<charT, traits> &seekg(off_type, ios_base::seekdir);

// Les opérations de gestion des manipulateurs :
    basic_istream<charT, traits> &operator>>
        (basic_istream<charT, traits> & (*pf)(
     basic_istream<charT, traits> &));
    basic_istream<charT, traits> &operator>>
        (basic_ios<charT, traits> & (*pf)(basic_ios<charT, traits> &));
    basic_istream<charT, traits> &operator>>
        (ios_base & (*pf)(ios_base &));
};
```

Tout comme la classe basic_ostream, le constructeur de la classe basic_istream prend en paramètre un pointeur sur l'objet gérant le tampon dans lequel les écritures devront être effectuées. Cependant, même s'il est possible de créer une instance de flux d'entrée simplement à l'aide de ce constructeur, cela n'est pas recommandé puisque la bibliothèque standard fournit des classes spécialisées permettant de créer des flux de sortie orientés fichiers ou chaînes de caractères.

L'utilisation des différentes surcharges de l'opérateur d'extraction des données formatées `operator>>` ne devrait pas poser de problème. Le compilateur détermine la surcharge à utiliser en fonction du type des données à lire, déterminé par la référence de variable fournie en paramètre. Cette surcharge récupère alors les informations dans le tampon associé au flux, les interprète et écrit la nouvelle valeur dans la variable.

Bien entendu, tout comme pour la classe basic_ostream, il est possible d'écrire de nouvelles surcharges de l'opérateur d'extraction afin de prendre en charge de nouveaux types de données. Idéalement, ces surcharges devront être également des fonctions `template` paramétrées par le type de caractère du flux sur lequel elles travaillent, et elles devront également utiliser une classe d'initialisation sentry. Cette classe a principalement pour but d'initialiser le flux d'entrée, éventuellement en le synchronisant avec un flux de sortie standard dont la classe basic_ostream peut être stockée dans le flux d'entrée à l'aide de la méthode `tie` de la classe de base basic_ios, et en supprimant les éventuels caractères blancs avant la lecture des données.

Notez que, contrairement à la classe sentry des flux de sortie, le constructeur de la classe sentry des flux d'entrée prend un deuxième paramètre. Ce paramètre est un booléen qui indique si les caractères blancs présents dans le flux de données doivent être éliminés avant l'opération de lecture ou non. En général, pour les opérations de lecture formatées, ce sont des caractères non significatifs et il faut effecivement supprimer ces caractères, aussi la valeur à spécifier pour ce second paramètre est-elle `false`. Comme c'est aussi la valeur par défaut, la manière d'utiliser de la classe sentry dans les opérateurs d'extraction est strictement identique à celle de la classe sentry des opérateurs d'insertion de la classe basic_ostream.

Exemple 15-7. Écriture d'un nouvel opérateur d'extraction pour un flux d'entrée

```
#include <iostream>
#include <string>
```

```
using namespace std;

// Définition d'un type de donnée privé :
struct Personne
{
    string Nom;
    string Prenom;
    int Age;       // En centimètres.
    int Taille;
};

// Définition de l'opérateur de lecture pour ce type :
template <class charT, class Traits>
basic_istream<charT, Traits> &operator>>(
    basic_istream<charT, Traits> &flux, Personne &p)
{
    // Inialisation du flux de sortie :
    typename basic_istream<charT, Traits>::sentry init(flux);
    if (init)
    {
        // Lecture du prénom et du nom :
        flux >> p.Prenom;
        flux >> p.Nom;
        // Lecture de l'âge :
        flux >> p.Age;
        // Lecture de la taille en mètres :
        double Taille;
        flux >> Taille;
        // Conversion en centimètres ;
        p.Taille = (int) (Taille * 100 + 0.5);
    }
    return flux;
}

int main(void)
{
    // Construit une nouvelle personne :
    Personne p;
    // Demande la saisie d'une personne :
    cout << "Prénom Nom Âge(ans) Taille(m) : ";
    cin >> p;
    // Affiche les valeurs lues :
    cout << endl;
    cout << "Valeurs saisies :" << endl;
    cout << p.Prenom << " " << p.Nom << " a " <<
        p.Age << " ans et mesure " <<
        p.Taille << " cm." << endl;
    return 0;
}
```

Note : La classe sentry est également utilisée par les méthodes de lecture de données non formatées. Pour ces méthodes, les caractères blancs sont importants et dans ce cas le second paramètre fourni au constructeur de la classe sentry est `true`.

Comme pour la classe sentry de la classe basic_ostream, l'utilisation de l'objet d'initialisation dans les tests est rendue possible par la présence de l'opérateur de transtypage vers le type bool. La valeur retournée est `true` si l'initialisation s'est bien faite et `false` dans le cas contraire.

> Remarquez également que le constructeur de la classe sentry est susceptible de lancer des exceptions selon la configuration du masque d'exceptions dans la classe de flux.

Les opérations de lecture de données non formatées sont un peu plus nombreuses pour les flux d'entrée que les opérations d'écriture non formatées pour les flux de sortie. En effet, la classe basic_istream donne non seulement la possibilité de lire un caractère simple ou une série de caractères, mais aussi de lire les données provenant du tampon de lecture et de les interpréter en tant que « lignes ». Une *ligne* est en réalité une série de caractères terminée par un caractère spécial que l'on nomme le *marqueur de fin de ligne*. En général, ce marqueur est le caractère '`\n`', mais il est possible de spécifier un autre caractère.

La lecture d'un caractère unique dans le flux d'entrée se fait à l'aide de la méthode `get`. Il existe deux surcharges de cette méthode, la première ne prenant aucun paramètre et renvoyant le caractère lu, et la deuxième prenant en paramètre une référence sur la variable devant recevoir le caractère lu et ne renvoyant rien. Ces deux méthodes extraient les caractères qu'elles lisent du tampon d'entrée que le flux utilise. Si l'on veut simplement lire la valeur du caractère suivant sans l'en extraire, il faut appeler la méthode `peek`. De plus, tout caractère extrait peut être réinséré dans le flux d'entrée (pourvu que le tampon sous-jacent accepte cette opération) à l'aide de l'une des deux méthodes `unget` ou `putback`. Cette dernière méthode prend en paramètre le caractère qui doit être réinséré dans le flux d'entrée. Notez que la réinsertion ne peut être réalisée que si le caractère fourni en paramètre est précisément le dernier caractère extrait.

Si l'on désire réaliser la lecture d'une série de caractères au lieu de les extraire un à un, il faut utiliser la méthode `read`. Cette méthode est la méthode de base pour les lectures non formatées puisqu'elle lit les données brutes de fonderie, sans les interpréter. Elle prend en paramètre un pointeur sur un tableau de caractères dans lequel les données seront écrites et le nombre de caractères à lire. Cette méthode ne vérifie pas la taille du tableau spécifié, aussi celui-ci doit-il être capable d'accueillir le nombre de caractères demandé. Il existe une variante de la méthode `read`, la méthode `readsome`, qui permet de lire les données présentes dans le tampon géré par le flux sans accéder au média que ce dernier prend en charge. Cette méthode prend également en paramètre un pointeur sur la zone mémoire devant recevoir les données et le nombre de caractères désiré, mais, contrairement à la méthode `read`, elle peut ne pas lire exactement ce nombre. En effet, la méthode `readsome` s'arrête dès que le tampon utilisé par le flux est vide, ce qui permet d'éviter les accès sur le périphérique auquel ce tampon donne accès. La méthode `readsome` renvoie le nombre de caractères effectivement lus.

Les méthodes de lecture des lignes sont à diviser en deux catégories. La première catégorie, constituée de plusieurs surcharges de la méthode `get`, permet d'effectuer une lecture des données du tampon jusqu'à ce que le tableau fourni en paramètre soit rempli ou qu'une fin de ligne soit atteinte. La deuxième catégorie de méthodes est constituée des surcharges de la méthode `getline`. Ces méthodes se distinguent des méthodes `get` par le fait qu'elles n'échouent pas lorsque la ligne lue (délimiteur de ligne compris) remplit complètement le tableau fourni en paramètre d'une part, et par le fait que le délimiteur de ligne est extrait du tampon d'entrée utilisé par le flux d'autre part. Autrement dit, si une ligne complète (c'est-à-dire avec son délimiteur) a une taille exactement égale à la taille du tableau fourni en paramètre, les méthodes `get` échoueront alors que les méthodes `getline` réussiront, car elles ne considèrent pas le délimiteur comme une information importante. Ceci revient à dire que les méthodes `getline` interprètent complètement le caractère délimiteur, alors que les méthodes `get` le traitent simplement comme le caractère auquel la lecture doit s'arrêter.

Dans tous les cas, un caractère nul terminal est inséré en lieu et place du délimiteur dans le tableau fourni en paramètre et devant recevoir les données. Comme le deuxième paramètre de ces méthodes indique la dimension de ce tableau, le nombre de caractères lu est au plus cette dimension moins un. Le nombre de caractères extraits du tampon d'entrée est quant à lui récupérable grâce à la méthode `gcount`. Remarquez que le caractère de fin de ligne est compté dans le nombre de caractères extraits

pour les méthodes `getline`, alors qu'il ne l'est pas pour les méthodes `get` puisque ces dernières ne l'extraient pas du tampon.

Enfin, il est possible de demander la lecture d'un certain nombre de caractères et de les passer sans en récupérer la valeur. Cette opération est réalisable à l'aide de la méthode `ignore`, qui ne prend donc pas de pointeurs sur la zone mémoire où les caractères lus doivent être stockés puisqu'ils sont ignorés. Cette méthode lit autant de caractères que spécifié, sauf si le caractère délimiteur indiqué en deuxième paramètre est rencontré. Dans ce cas, ce caractère est extrait du tampon d'entrée, ce qui fait que la méthode `ignore` se comporte exactement comme les méthodes `getline`.

Exemple 15-8. Lectures de lignes sur le flux d'entrée standard

```
#include <iostream>
#include <sstream>

using namespace std;

int main(void)
{
    // Tableau devant recevoir une ligne :
    char petit_tableau[10];
    // Lit une ligne de 9 caractères :
    cout << "Saisissez une ligne :" << endl;
    cin.getline(petit_tableau, 10);
    if (cin.fail())
        cout << "Ligne trop longue !" << endl;
    cout << "Lu : ***" << petit_tableau << "***" << endl;
    // Lit une ligne de taille arbitraire via un tampon :
    cout << "Saisissez une autre ligne :" << endl;
    stringbuf s;
    cin.get(s);
    // Affiche la ligne lue :
    cout << "Lu : ***" << s.str() << "***";
    // Extrait le caractère de saut de ligne
    // et ajoute-le au flux de sortie standard :
    cout << (char) cin.get();
    return 0;
}
```

Note : Remarquez que le caractère de saut de ligne étant lu, il est nécessaire de saisir deux retours de chariot successifs pour que la méthode `getline` renvoie son résultat. Comme pour toutes les méthodes de lectures formatées, ce caractère interrompt la lecture dans le flux d'entrée standard du programme et se trouve donc encore dans le tampon d'entrée lors de la lecture suivante. Cela explique que dans le cas de lectures successives, il faut extraire ce caractère du flux d'entrée manuellement, par exemple à l'aide de la méthode `get`. C'est ce que cet exemple réalise sur sa dernière ligne pour l'envoyer sur le flux de sortie standard.

De plus, on ne peut pas prévoir, a priori, quelle sera la taille des lignes saisies par l'utilisateur. On ne procédera donc pas comme indiqué dans cet exemple pour effectuer la lecture de lignes en pratique. Il est en effet plus facile d'utiliser la fonction `getline`, que l'on a décrit dans la Section 14.1.8 dans le cadre du type basic_string. En effet, cette fonction permet de lire une ligne complète sans avoir à se soucier de sa longueur maximale et de stocker le résultat dans une basic_string.

La classe basic_istream dispose également de méthodes permettant de manipuler le tampon qu'elle utilise pour lire de nouvelles données. La méthode `sync` permet de synchroniser le tampon d'entrée avec le média auquel il donne accès, puisqu'elle appelle la méthode `pubsync` de ce tampon. Pour les flux d'entrée, cela n'a pas réellement d'importance parce que l'on ne peut pas écrire dedans. La méthode `tellg` permet de déterminer la position du pointeur de lecture courant, et les deux surcharges de la méthode `seekg` permettent de repositionner ce pointeur. Nous verrons un exemple d'utilisation de ces méthodes dans la description des classes de flux pour les fichiers.

Enfin, les flux d'entrée disposent également de quelques manipulateurs permettant de les configurer simplement à l'aide de l'opérateur `operator>>`. Ces manipulateurs sont présentés dans le tableau ci-dessous :

Tableau 15-7. Manipulateurs des flux d'entrée

Manipulateur	Fonction
`boolalpha`	Active l'interprétation des booléens sous forme de textuelle.
`noboolalpha`	Désactive l'interprétation des booléens sous forme textuelle.
`hex`	Utilise la base 16 pour l'interprétation des nombres entiers.
`oct`	Utilise la base 8 pour l'interprétation des nombres entiers.
`dec`	Utilise la base 10 pour l'interprétation des nombres entiers.
`skipws`	Ignore les espaces lors des entrées formatées.
`noskipws`	Conserve les espaces lors des entrées formatées.
`ws`	Supprime tous les espaces présents dans le flux d'entrée jusqu'au premier caractère non blanc.

Ces manipulateurs s'utilisent directement à l'aide de l'opérateur `operator>>`, exactement comme les manipulateurs de la classe basic_ostream s'utilisent avec l'opérateur d'insertion normal.

15.4.3. La classe basic_iostream

La bibliothèque standard définit dans l'en-tête `iostream` la classe `template` basic_iostream afin de permettre à la fois les opérations d'écriture et les opérations de lecture sur les flux. En fait, cette classe n'est rien d'autre qu'une classe dérivée des deux classes basic_ostream et basic_istream qui fournissent respectivement, comme on l'a vu, toutes les fonctionnalités de lecture et d'écriture sur un tampon.

La classe basic_iostream ne comporte pas d'autres méthodes qu'un constructeur et un destructeur, qui servent uniquement à initialiser et à détruire les classes de base basic_ostream et basic_istream. L'utilisation de cette classe ne doit donc pas poser de problème particulier et je vous invite à vous référer aux descriptions des classes de base si besoin est.

> **Note :** Tout comme ses classes de base, la classe basic_iostream sera rarement utilisée directement. En effet, elle dispose de classes dérivées spécialisées dans les opérations d'écriture et de lecture sur fichiers ou dans des chaînes de caractères, classes que l'on présentera dans les sections suivantes. Ce sont ces classes que l'on utilisera en pratique lorsque l'on désirera créer un nouveau flux pour lire et écrire dans un fichier ou dans une basic_string.
>
> Vous aurez peut-être remarqué que les classes basic_ostream et basic_istream utilisent un héritage virtuel pour récupérer les fonctionnalités de la classe de base basic_ios. La raison en est que la classe basic_iostream réalise un héritage multiple sur ses deux classes de base et que les données de la classe basic_ios ne doivent être présente qu'en un seul exemplaire dans les flux d'entrée / sortie. Cela implique que les constructeurs des classes dérivées de la classe

basic_iostream prenant des paramètres doivent appeler explicitement les constructeurs de toutes leur classes de base. Voyez la Section 7.6 pour plus de détails sur les notions d'héritage multiple et de classes virtuelles.

15.5. Les flux d'entrée / sortie sur chaînes de caractères

Afin de donner la possibilité aux programmeurs d'effectuer les opérations de formatage des données en mémoire aussi simplement qu'avec les classes de gestion des flux d'entrée / sortie standards, la bibliothèque d'entrée / sortie définit trois classes de flux capables de travailler dans des chaînes de caractères de type basic_string. Ces classes sont les classes basic_ostringstream, pour les écritures dans les chaînes de caractères, basic_istringstream, pour les lectures de données stockées dans les chaînes de caractères, et basic_stringstream, pour les opérations à la fois d'écriture et de lecture.

Ces classes dérivent respectivement des classes de flux basic_istream, basic_ostream et basic_iostream et reprennent donc à leur compte toutes les fonctions de formatage et d'écriture de ces classes. Les écritures et les lectures de données en mémoire se font donc, grâce à ces classes, aussi facilement qu'avec les flux d'entrée / sortie standards, et ce de manière complètement transparente.

En fait, les classes de flux orientées chaînes de caractères fonctionnent exactement comme leurs classes de base, car toutes les fonctionnalités de gestion des chaînes de caractères sont encapsulées au niveau des classes de gestion des tampons qui ont été présentées au début de ce chapitre. Cependant, elles disposent de méthodes spécifiques qui permettent de manipuler les chaînes de caractères sur lesquelles elles travaillent. Par exemple, la classe basic_ostringstream est déclarée comme suit dans l'en-tête `sstream` :

```
template <class charT,
    class traits = char_traits<charT>,
    class Allocator = allocator<charT> >
class basic_ostringstream : public basic_ostream<charT, traits>
{
public:
// Les types de données :
    typedef charT            char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

// Les constructeurs et destructeurs :
    explicit basic_ostringstream(ios_base::openmode mode = ios_base::out);
    explicit basic_ostringstream(
        const basic_string<charT, traits, Allocator> &chaine,
        ios_base::openmode mode = ios_base::out);
    virtual ~basic_ostringstream();

// Les méthodes de gestion de la chaîne de caractères :
    basic_stringbuf<charT, traits, Allocator> *rdbuf() const;
    basic_string<charT, traits, Allocator> str() const;
    void    str(const basic_string<charT, traits, Allocator> &chaine);
};
```

Les classes basic_istringstream et basic_stringstream sont déclarées de manière identique, à ceci près que les classes de base et les valeurs par défaut pour les modes d'ouverture du tampon sur la chaîne de caractères ne sont pas les mêmes.

Comme vous pouvez le constater, il est possible de construire un flux d'entrée / sortie sur une chaîne de caractères de différentes manières. La première méthode est de construire un objet flux, puis de préciser, pour les flux d'entrée, la chaîne de caractères dans laquelle les données à lire se trouvent à l'aide de la méthode `str`. La deuxième méthode est tout simplement de fournir tous les paramètres en une seule fois au constructeur. En général, les valeurs par défaut spécifiées dans les constructeurs correspondent à ce que l'on veut faire avec les flux, ce qui fait que la construction de ces flux est extrêmement simple.

Une fois construit, il est possible de réaliser toutes les opérations que l'on désire sur le flux. Dans le cas des flux d'entrée, il est nécessaire que le flux ait été initialisé avec une chaîne de caractères contenant les données à lire, et l'on ne cherche généralement pas à récupérer cette chaîne après usage du flux. Pour les flux de sortie, cette initialisation est inutile. En revanche, le résultat des opérations de formatage sera généralement récupéré à l'aide de la méthode `str` une fois celles-ci réalisées.

Exemple 15-9. Utilisation de flux d'entrée / sortie sur chaînes de caractères

```
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

int main(void)
{
    // Lit une ligne en entrée :
    cout << "Entier Réel Chaîne : ";
    string input;
    getline(cin, input);
    // Interprète la ligne lue :
    istringstream is(input);
    int i;
    double d;
    string s;
    is >> i >> d;
    is >> ws;
    getline(is, s);
    // Formate la réponse :
    ostringstream os;
    os << "La réponse est : " << endl;
    os << s << " " << 2*i << " " << 2*d << endl;
    // Affiche la chaîne de la réponse :
    cout << os.str();
    return 0;
}
```

Comme l'exemple précédent vous le montre, l'utilisation des flux d'entrée / sortie de la bibliothèque standard sur les chaînes de caractères est réellement aisée. Comme ces opérations ne peuvent être réalisées qu'en mémoire, c'est à dire en dehors du contexte du système d'exploitation utilisé, les classes de flux de la bibliothèque restent utiles même si les opérations d'entrée / sortie du système se font de telle manière que les objets `cin` et `cout` ne sont pratiquement pas utilisables.

15.6. Les flux d'entrée / sortie sur fichiers

Les classes d'entrée / sortie sur les fichiers sont implémentées de manière similaire aux classes d'entrée / sortie sur les chaînes de caractères, à ceci près que leurs méthodes spécifiques permettent de manipuler un fichier au lieu d'une chaîne de caractères. Ainsi, la classe basic_ofstream dérive de basic_ostream, la classe basic_ifstream de la classe basic_istream, et la classe basic_fstream de la classe basic_iostream. Toutes ces classes sont déclarées dans l'en-tête `fstream`. Vous trouverez à titre d'exemple la déclaration de la classe basic_ofstream tel qu'il apparaît dans cet en-tête :

```
template <class charT,
    class traits = char_traits<charT> >
class basic_ofstream : public basic_ostream<charT, traits>
{
public:
// Les types de données :
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

// Les constructeurs et destructeurs :
    basic_ofstream();
    explicit basic_ofstream(const char *nom,
        ios_base::openmode mode = ios_base::out | ios_base::trunc);

// Les méthodes de gestion du fichier :
    basic_filebuf<charT, traits> *rdbuf() const;
    bool is_open();
    void open(const char *nom, ios_base::openmode mode = out | trunc);
    void close();
};
```

Comme pour les flux d'entrée / sortie sur les chaînes de caractères, il est possible d'initialiser le flux dès sa construction ou a posteriori. Les méthodes importantes sont bien entendu la méthode `open`, qui permet d'ouvrir un fichier, la méthode `is_open`, qui permet de savoir si le flux contient déjà un fichier ouvert ou non, et la méthode `close`, qui permet de fermer le fichier ouvert.

Exemple 15-10. Utilisation de flux d'entrée / sortie sur un fichier

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main(void)
{
    // Lit les données :
    int i;
    double d, e;
    cout << "Entier Réel Réel : ";
    cin >> i >> d >> e;
    // Enregistre ces données dans un fichier :
    ofstream f("fichier.txt");
    if (f.is_open())
```

```
    {
        f << "Les données lues sont : " <<
            i << " " << d << " " << e << endl;
        f.close();
    }
    return 0;
}
```

Note : Il est important de bien noter que le destructeur des flux ne ferme pas les fichiers. En effet, ce sont les tampons utilisés de manière sous-jacente par les flux qui réalisent les opérations sur les fichiers. Il est même tout à fait possible d'accéder à un même fichier avec plusieurs classes de flux, bien que cela ne soit pas franchement recommandé. Vous devrez donc prendre en charge vous-même les opérations d'ouverture et de fermeture des fichiers.

Bien entendu, les classes de flux permettant d'accéder à des fichiers héritent des méthodes de positionnement de leurs classes de base. Ainsi, les classes de lecture dans un fichier disposent des méthodes `tellg` et `seekg`, et les classes d'écriture disposent des méthodes `tellp` et `seekp`. Ces opérations permettent respectivement de lire et de fixer une nouvelle valeur du pointeur de position du fichier courant.

Exemple 15-11. Repositionnement du pointeur de fichier dans un flux d'entrée / sortie

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main(void)
{
    // Ouvre le fichier de données :
    fstream f("fichier.txt",
        ios_base::in | ios_base::out | ios_base::trunc);
    if (f.is_open())
    {
        // Écrit les données :
        f << 2 << " " <<  45.32 << " " << 6.37 << endl;
        // Replace le pointeur de fichier au début :
        f.seekg(0);
        // Lit les données :
        int i;
        double d, e;
        f >> i >> d >> e;
        cout << "Les données lues sont : " <<
            i << " " << d << " " << e << endl;
        // Ferme le fichier :
        f.close();
    }
    return 0;
}
```

Note : Les classes d'entrée / sortie sur fichier n'utilisent qu'un seul tampon pour accéder aux fichiers. Par conséquent, il n'existe qu'une seule position dans le fichier, qui sert à la fois à la lecture et à l'écriture.

Chapitre 16. Les locales

Il existe de nombreux alphabets et de nombreuses manières d'écrire les nombres, les dates et les montants de part le monde. Chaque pays, chaque culture dispose en effet de ses propres conventions et de ses propres règles, et ce dans de nombreux domaines. Par exemple, les Anglo-saxons ont pour coutume d'utiliser le point (caractère '`.`') pour séparer les unités de la virgule lorsqu'ils écrivent des nombres à virgule et d'utiliser une virgule (caractère '`,`') entre chaque groupe de trois chiffres pour séparer les milliers des millions, les millions des milliards, etc. En France, c'est la virgule est utilisée pour séparer les unités de la partie fractionnaire des nombres à virgule, et le séparateur des milliers est simplement un espace. De même, ils ont l'habitude d'écrire les dates en mettant le mois avant les jours, alors que les Français font l'inverse.

Il va de soi que ce genre de différences rend techniquement très difficile l'internationalisation des programmes. Une solution est tout simplement de dire que les programmes travaillent dans une langue « neutre », ce qui en pratique revient souvent à dire l'anglais puisque c'est la langue historiquement la plus utilisée en informatique. Hélas, si cela convenait parfaitement aux programmeurs, ce ne serait certainement pas le cas des utilisateurs ! Il faut donc, à un moment donné ou à un autre, que les programmes prennent en compte les conventions locales de chaque pays ou de chaque peuple.

Ces conventions sont extrêmement nombreuses et portent sur des domaines aussi divers et variés que la manière d'écrire les nombres, les dates ou de coder les caractères et de classer les mots dans un dictionnaire. En informatique, il est courant d'appeler l'ensemble des conventions d'un pays la *locale* de ce pays. Les programmes qui prennent en compte la locale sont donc dits *localisés* et sont capables de s'adapter aux préférences nationales de l'utilisateur.

> **Note :** Le fait d'être localisé ne signifie pas pour autant pour un programme que tous ses messages sont traduits dans la langue de l'utilisateur. La localisation ne prend en compte que les aspects concernant l'écriture des nombres et les alphabets utilisés. Afin de bien faire cette distinction, on dit que les programmes capables de communiquer avec l'utilisateur dans sa langue sont *internationalisés*. La conversion d'un programme d'un pays à un autre nécessite donc à la fois la localisation de ce programme et son internationalisation.

Si la traduction de tous les messages d'un programme ne peut pas être réalisée automatiquement, il est toutefois possible de prendre en compte les locales relativement facilement. En effet, les fonctionnalités des bibliothèques C et C++, en particulier les fonctionnalités d'entrée / sortie, peuvent généralement être paramétrées par la locale de l'utilisateur. La gestion des locales est donc complètement prise en charge par ces bibliothèques et un même programme peut donc être utilisé sans modification dans divers pays.

> **Note :** En revanche, la traduction des messages ne peut bien évidemment pas être prise en charge par la bibliothèque standard, sauf éventuellement pour les messages d'erreur du système. La réalisation d'un programme international nécessite donc de prendre des mesures particulières pour faciliter la traduction de ces messages. En général, ces mesures consistent à isoler les messages dans des modules spécifiques et à ne pas les utiliser directement dans le code du programme. Ainsi, il suffit simplement de traduire les messages de ces modules pour ajouter le support d'une nouvelle langue à un programme existant. Le code source n'a ainsi pas à être touché, ce qui limite les risques d'erreurs.

La gestion des locales en C++ se fait par l'intermédiaire d'une classe générale, la classe locale, qui permet de stocker tous les paramètres locaux des pays. Cette classe est bien entendu utilisée par les flux d'entrée / sortie de la bibliothèque standard, ce qui fait que vous n'aurez généralement qu'à

initialiser cette classe au début de vos programmes pour leur faire prendre en compte les locales. Cependant, il se peut que vous ayez à manipuler vous-même des locales ou à définir de nouvelles conventions nationales, surtout si vous écrivez des surcharges des opérateurs de formatage des flux `operator<<` et `operator>>`. Ce chapitre présente donc les notions générales des locales, les différentes classes mises en œuvre au sein d'une même locale pour prendre en charge tous les aspects de la localisation, et la manière de définir ou de redéfinir un de ces aspects afin de compléter une locale existante.

16.1. Notions de base et principe de fonctionnement des facettes

Comme il l'a été dit plus haut, les locales comprennent différents aspects qui traitent chacun d'une des conventions nationales de la locale. Par exemple, la manière d'écrire les nombres constitue un de ces aspects, tout comme la manière de classer les caractères ou la manière d'écrire les heures et les dates.

Chacun de ces aspects constitue ce que la bibliothèque standard C++ appelle une *facette*. Les facettes sont gérées par des classes C++, dont les méthodes permettent d'obtenir les informations spécifiques aux données qu'elles manipulent. Certaines facettes fournissent également des fonctions permettant de formater et d'interpréter ces données en tenant compte des conventions de leur locale. Chaque facette est identifiée de manière unique dans le programme, et chaque locale contient une collection de facettes décrivant tous ses aspects.

La bibliothèque standard C++ fournit bien entendu un certain nombre de facettes prédéfinies. Ces facettes sont regroupées en *catégories* qui permettent de les classer en fonction du type des opérations qu'elles permettent de réaliser. La bibliothèque standard définit six catégories de facettes, auxquelles correspondent les valeurs de six constantes de la classe locale :

- la catégorie `ctype`, qui regroupe toutes les facettes permettant de classifier les caractères et de les convertir d'un jeu de caractère en un autre ;
- la catégorie `collate`, qui comprend une unique facette permettant de comparer les chaînes de caractères en tenant compte des caractères de la locale courante et de la manière de les utiliser dans les classements alphabétiques ;
- la catégorie `numeric`, qui comprend toutes les facettes prenant en charge le formatage des nombres ;
- la catégorie `monetary`, qui comprend les facettes permettant de déterminer les symboles monétaires et la manière d'écrire les montants ;
- la catégorie `time`, qui comprend les facettes capables d'effectuer le formatage et l'écriture des dates et des heures ;
- la catégorie `message`, qui contient une unique facette permettant de faciliter l'internationalisation des programmes en traduisant les messages destinés aux utilisateurs.

Bien entendu, il est possible de définir de nouvelles facettes et de les inclure dans une locale existante. Ces facettes ne seront évidemment pas utilisées par les fonctions de la bibliothèque standard, mais le programme peut les récupérer et les utiliser de la même manière que les facettes standards.

Les mécanismes de définition, d'identification et de récupération des facettes, sont tous pris en charge au niveau de la classe locale. La déclaration de cette classe est réalisée de la manière suivante dans l'en-tête `locale` :

```
class locale
{
public:
// Les types de données :
    // Les catégories de facettes :
    typedef int category;
    static const category   // Les valeurs de ces constantes
    {                       // sont spécifiques à chaque implémentation
        none     = VAL0,
        ctype    = VAL1, collate  = VAL2,
        numeric  = VAL3, monetary = VAL4,
        time     = VAL5, messages = VAL6,
        all = collate | ctype | monetary | numeric | time  | messages;
    };

    // La classe de base des facettes :
    class facet
    {
    protected:
        explicit facet(size_t refs = 0);
        virtual ~facet();
    private:
        // Ces méthodes sont déclarées mais non définies :
        facet(const facet &);
        void operator=(const facet &);
    };

    // La classe d'identification des facettes :
    class id
    {
    public:
        id();
    private:
        // Ces méthodes sont déclarées mais non définies :
 void id(const id &);
 void operator=(const id &);
    };

// Les constructeurs :
    locale() throw()
    explicit locale(const char *nom);
    locale(const locale &) throw();
    locale(const locale &init, const char *nom, category c);
    locale(const locale &init1, const locale &init2, category c);

    template <class Facet>
    locale(const locale &init, Facet *f);

    template <class Facet>
    locale(const locale &init1, const locale &init2);

// Le destructeur :
    ~locale() throw();

// Opérateur d'affectation :
    const locale &operator=(const locale &source) throw();
```

```
// Les méthodes de manipulation des locales :
    basic_string<char> name() const;
    bool operator==(const locale &) const;
    bool operator!=(const locale &) const;
    template <class charT, class Traits, class Allocator>
    bool operator()(
        const basic_string<charT, Traits, Allocator> &s1,
        const basic_string<charT, Traits, Allocator> &s2) const;

// Les méthodes de sélection des locales :
    static       locale  global(const locale &);
    static const locale &classic();
};
```

Comme vous pouvez le constater, outre les constructeurs, destructeur et méthodes générales, la classe locale contient la déclaration de deux sous-classes utilisées pour la définition des facettes : la classe facet et la classe id.

La classe facet est la classe de base de toutes les facettes. Son rôle est essentiellement d'éviter que l'on puisse dupliquer une facette ou en copier une, ce qui est réalisé en déclarant en zone privée le constructeur de copie et l'opérateur d'affectation. Comme ces méthodes ne doivent pas être utilisées, elles ne sont pas définies non plus, seule la déclaration est fournie par la bibliothèque standard. Notez que cela n'est pas dérangeant pour l'utilisation de la classe facet, puisque comme ces méthodes ne sont pas virtuelles et qu'elles ne seront jamais utilisées dans les programmes, l'éditeur de liens ne cherchera pas à les localiser dans les fichiers objets du programme. Ainsi, les instances de facettes ne peuvent être ni copiées, ni dupliquées. En fait, les facettes sont destinées à être utilisées au sein des locales, qui prennent en charge la gestion de leur durée de vie. Toutefois, si l'on désire gérer soi-même la durée de vie d'une facette, il est possible de le signaler lors de la construction de la facette. Le constructeur de base de la classe facet prend en effet un paramètre unique qui indique si la durée de vie de la facette doit être prise en charge par la locale dans laquelle elle se trouve ou si elle devra être explicitement détruite par le programmeur (auquel cas ce paramètre doit être fixé à `1`). En général, la valeur par défaut `0` convient dans la majorité des cas et il n'est pas nécessaire de fournir de paramètre au constructeur des facettes.

La classe id quant à elle est utilisée pour définir des identifiants uniques pour chaque classe de facette. Ces identifiants permettent à la bibliothèque standard de distinguer les facettes les unes des autres, à l'aide d'une clef unique dont le format n'est pas spécifié, mais qui est déterminée par la classe id automatiquement lors de la première création de chaque facette. Cet identifiant est en particulier utilisé pour retrouver les facettes dans la collection des facettes gérées par la classe locale.

Pour que ce mécanisme d'enregistrement fonctionne, il faut que chaque classe de facette définisse une donnée membre statique *`id`* en zone publique, dont le type est la sous-classe id de la classe locale. Cette donnée membre étant statique, elle appartient à la classe et non à chaque instance, et permet donc bien d'identifier chaque classe de facette. Lors du chargement du programme, les variables statiques sont initialisées à 0, ce qui fait que les facettes disposent toutes d'un identifiant nul. Ceci permet aux méthodes de la bibliothèque standard de déterminer si un identifiant a déjà été attribué à la classe d'une facette ou non lorsqu'elle est utilisée pour la première fois. Si c'est le cas, cet identifiant est utilisé tel quel pour référencer cette classe, sinon, il est généré automatiquement et stocké dans la donnée membre *`id`* de la classe.

Ainsi, pour définir une facette, il suffit simplement d'écrire une classe dérivant de la classe locale::facet et contenant une donnée membre statique et publique *`id`* de type locale::id. Dès lors, cette facette se verra attribuer automatiquement un identifiant unique, qui permettra d'utiliser les fonctions de recherche de facettes dans les locales. Ces fonctions utilisent bien entendu la donnée membre *`id`*

du type de la facette à rechercher et s'en servent en tant qu'index dans la collection des facettes de la locale à utiliser. La manière de réaliser ces opérations n'est pas décrite par la norme du C++, mais le principe est là.

Les fonctions de recherche des facettes sont également déclarées dans l'en-tête `locale`. Ce sont des fonctions `template` paramétrées par le type des facettes, qui prennent en paramètre la locale dans laquelle la facette doit être recherchée :

```
template <class Facet>
const Facet &use_facet(const locale &);

template <class Facet>
bool        has_facet(const locale &);
```

La fonction `use_facet` permet de récupérer l'instance d'une facette dans la locale passée en paramètre. Comme la signature de cette fonction `template` ne permet pas de déterminer le type de la facette, et donc l'instance à utiliser pour l'appel, il est nécessaire de spécifier explicitement ce type entre crochets après le nom de la fonction. Par exemple, pour récupérer la facette num_put<char> de la locale classique de la bibliothèque C, on fera l'appel suivant :

```
use_facet<num_put<char> >(locale::classic());
```

Les facettes fournies par la bibliothèque standard sont généralement disponibles et peuvent être utilisées avec les locales. Les méthodes spécifiques à chacune de ces facettes peuvent donc être appelées sur la référence de la facette retournée par la fonction `use_facet`. En revanche, les facettes définies par l'utilisateur peuvent ne pas être présentes dans la locale fournie en paramètre à la fonction `use_facet`. Dans ce cas, cette fonction lance une exception de type bad_cast. Comme il peut être utile en certaines circonstances de déterminer dynamiquement si une locale contient ou non une facette, la bibliothèque standard met à disposition la fonction globale `has_facet`. Cette fonction s'utilise de la même manière que la fonction `use_facet`, mais au lieu de renvoyer la facette demandée, elle retourne un booléen indiquant si la locale fournie en paramètre contient ou non cette facette.

Les programmes peuvent créer plusieurs locales afin de prendre en compte plusieurs jeux de paramètres internationaux s'ils le désirent, mais ils doivent dans ce cas manipuler ces locales eux-mêmes dans toutes les opérations susceptibles d'utiliser la notion de locale. Par exemple, ils doivent spécifier la locale à utiliser avant chaque opération d'entrée / sortie en appelant la méthode `imbue` des flux utilisés. Comme cela n'est pas très pratique, la bibliothèque standard définit une locale globale, qui est la locale utilisée par défaut lorsqu'un programme ne désire pas spécifier explicitement la locale à utiliser. Cette locale peut être récupérée à tout moment en créant un simple objet de type locale, en utilisant le constructeur par défaut de la classe locale. Ce constructeur initialise en effet la locale en cours de construction avec tous les paramètres de la locale globale. Ainsi, pour récupérer la facette num_put<char> de la locale globale, on fera l'appel suivant :

```
use_facet<num_put<char> >(locale());
```

Vous remarquerez que la locale fournie en paramètre à la fonction `use_facet` n'est plus, contrairement à l'exemple précédent, la locale renvoyée par la méthode statique `classic` de la classe locale, mais une copie de la locale globale.

Il est possible de construire une locale spécifique explicitement avec le constructeur de la classe locale qui prend le nom de la locale à utiliser en paramètre. Ce nom peut être l'un des noms standards

`"C"`, `""`, ou toute autre valeur dont la signification n'est pas normalisée. Le nom de locale vide (`""`) permet de construire une locale dont les paramètres sont initialisés en fonction de l'environnement d'exécution du programme. C'est donc la valeur que l'on utilisera en général, car cela permet de paramétrer le comportement des programmes facilement, sans avoir à les modifier et à les recompiler.

> **Note :** La manière de définir la locale dans l'environnement d'exécution des programmes est spécifique à chaque système d'exploitation et n'est normalisé ni par la norme C++, ni par la norme C. La norme POSIX précise cependant que cela peut être réalisé par l'intermédiaire de variables d'environnement. Par exemple, si la variable d'environnement LANG n'est pas définie, la locale utilisée sera la locale de la bibliothèque C. En revanche, si cette variable d'environnement contient la valeur `"fr_FR"`, la locale utilisée sera celle des francophones de France. Les formats des nombres, des dates, etc. utilisés seront donc ceux qui sont en vigueur en France.

Les autres constructeurs de la classe locale permettent de créer de nouvelles locales en recopiant les facettes d'une locale existante, éventuellement en ajoutant de nouvelles facettes non standards ou en redéfinissant certaines facettes de la locale modèle. Bien entendu, le constructeur de copie recopie toutes les facettes de la locale source dans la locale en cours de construction. Les deux constructeurs suivants permettent de recopier toutes les facettes d'une locale, sauf les facettes identifiées par la catégorie spécifiée en troisième paramètre. Pour cette catégorie, les facettes utilisées sont celles de la locale spécifiée en deuxième paramètre. Il est possible ici d'identifier cette deuxième locale soit par son nom, soit directement par l'intermédiaire d'une référence. Enfin, les deux constructeurs `template` permettent de créer une locale dont toutes les facettes sont initialisées à partir d'une locale fournie en paramètre, sauf la facette dont le type est utilisé en tant que paramètre `template`. Pour cette facette, la valeur à utiliser peut être spécifiée directement en deuxième paramètre ou extraite d'une autre locale, elle aussi spécifiée en deuxième paramètre. Nous verrons plus en détail dans la Section 16.3 la manière de procéder pour insérer une nouvelle facette ou remplacer une facette existante.

Enfin, la classe locale dispose de deux méthodes statiques permettant de manipuler les locales du programme. La méthode `classic` que l'on a utilisée dans l'un des exemples précédents permet d'obtenir la locale représentant les options de la bibliothèque C standard. Cette locale est la locale utilisée par défaut par les programmes qui ne définissent pas les locales utilisées. La méthode `global` quant à elle permet de spécifier la locale globale du programme. Cette méthode prend en paramètre un objet de type locale à partir duquel la locale globale est initialisée. Il est donc courant de faire un appel à cette méthode dès le début des programmes C++.

Exemple 16-1. Programme C++ prenant en compte la locale de l'environnement

```
#include <ctime>
#include <iostream>
#include <locale>

using namespace std;

int main(void)
{
    // Utilise la locale définie dans l'environnement
    // d'exécution du programme :
    locale::global(locale(""));
    // Affiche la date courante :
    time_t date;
    time(&date);
    struct tm *TL = localtime(&date);
```

```
    use_facet<time_put<char> >(locale()).put(
        cout, cout, ' ', TL, 'c');
    cout << endl;
    // Affiche la date avec la fonction strftime
    // de la bibliothèque C :
    char s[64];
    strftime(s, 64, "%c", TL);
    cout << s << endl;
    return 0;
}
```

La méthode `global` de la classe global appelle automatiquement la méthode `setlocale` si la locale fournie en paramètre a un nom. Cela signifie que les locales de la bibliothèque standard C++ et celles de la bibliothèque standard C sont compatibles et utilisent les mêmes conventions de nommage. En particulier, les programmes qui veulent utiliser la locale définie dans leur environnement d'exécution peuvent utiliser la locale anonyme `""`. C'est ce que fait le programme de l'exemple précédent, qui affiche la date au format de la locale définie par l'utilisateur en passant par les mécanismes du C++ (via la facette time_put, qui sera décrite en détail dans la Section 16.2.6) et par la fonction `strftime` de la bibliothèque C.

> **Note :** Les fonctions `time`, `localtime` et `strftime` sont des fonctions de la bibliothèque C standard. Elles permettent respectivement d'obtenir une valeur de type time_t représentant la date courante, de la convertir en une structure contenant les différentes composantes de la date en temps local, et de formater cette date selon les conventions de la locale courante. Ces fonctions ne seront pas décrites plus en détail ici. Vous pouvez consulter la bibliographie si vous désirez obtenir plus de détails sur la bibliothèque C et les fonctions qu'elle contient.

16.2. Les facettes standards

Cette section présente l'ensemble des facettes standards définies par la bibliothèque standard. La première partie décrit l'architecture générale à laquelle les facettes standards se conforment, et les parties suivantes donnent une description des principales fonctionnalités fournies par chacune des catégories de facettes.

16.2.1. Généralités

Les facettes fournies par la bibliothèque standard sont des classes `template` paramétrées par le type des caractères sur lesquels elles travaillent. Pour quelques-unes de ces facettes, la bibliothèque standard définit une spécialisation pour les types char ou wchar_t, spécialisation dont le but est d'optimiser les traitements des facettes pour les flux d'entrée / sortie standards sur ces types.

Certaines de ces facettes ne sont utilisées que pour fournir des informations aux autres parties de la bibliothèque standard C++. D'autres, en revanche, permettent de réaliser les opérations de formatage et d'analyse syntaxique sur lesquelles les flux d'entrée / sortie s'appuient pour implémenter les opérateurs `operator<<` et `operator>>`. Ces facettes disposent alors de méthodes `put` et `get` qui permettent d'effectuer ces deux types d'opération.

Les traitements effectués par les facettes doivent prendre en compte les paramètres de leur locale ainsi que les options de formatage stockées dans les flux sur lesquelles les entrées / sorties doivent être effectuées. Pour cela, les facettes doivent disposer d'un moyen de récupérer la locale dont elles font

partie ou dont elles doivent utiliser les paramètres. Généralement, les facettes qui réalisent les opérations d'entrée / sortie pour le compte des flux utilisent la méthode `getloc` de ces derniers pour obtenir la locale dont elles doivent utiliser les paramètres. Les autres facettes ne peuvent pas procéder de la même manière, car elles ne disposent pas forcément d'un objet flux pour déterminer la locale courante. Afin de résoudre ce problème, la bibliothèque standard définit des classes de facettes dérivées et dont le constructeur prend en paramètre le nom de la locale à laquelle ces facettes appartiennent. Ces classes sont donc initialisées, dès leur construction, avec le nom de la locale dans laquelle elles se trouvent, ce qui leur permet éventuellement d'effectuer des traitements dépendants de cette locale. Les noms de ces classes de facettes dérivées sont les mêmes que ceux de leurs classes de base, à ceci près qu'ils sont suffixés par la chaîne « `_byname` ». Par exemple, la facette ctype, qui, comme on le verra plus loin, permet de classer les caractères selon leur nature, dispose d'une classe dérivée ctype_byname dont le constructeur prend en paramètre le nom de la locale dont la facette fait partie.

> **Note :** Les implémentations de la bibliothèque standard fournies avec les environnements de développement C++ ne sont pas tenues de fournir ces facettes pour chaque locale existante dans le monde. En réalité, quasiment aucun environnement ne le fait à l'heure actuelle. En revanche, toutes les facettes standards doivent au moins être fournies et fonctionner correctement avec les locales `"C"` et `""`.

Les facettes sont écrites de telle manière qu'elles peuvent facilement être remplacées par des facettes plus spécifiques. Ainsi, leurs méthodes publiques appellent toutes des méthodes virtuelles, qui peuvent parfaitement être redéfinies par des classes dérivées désirant remplacer l'un des traitements effectués par la bibliothèque standard.

Généralement, les noms des méthodes virtuelles sont les mêmes que ceux des méthodes publiques qui les utilisent, précédés du préfixe « `do_` ». Par exemple, si une facette fournit une méthode publique nommée `put`, la méthode virtuelle appelée par celle-ci se nommera `do_put`. La manière de redéfinir les méthodes d'une facette existante et de remplacer cette facette par une de ses classes dérivées dans une locale sera décrite en détail dans la Section 16.3.2.

16.2.2. Les facettes de manipulation des caractères

La bibliothèque standard définit deux facettes permettant de manipuler les caractères. La première facette, la facette ctype, fournit les fonctions permettant de classer les caractères en différentes catégories. Ces catégories comprennent les lettres, les chiffres, les caractères imprimables, les caractères graphiques, etc. La deuxième facette permet quant à elle d'effectuer les conversions entre les différents types existants d'encodage de caractères. Il s'agit de la facette code_cvt.

16.2.2.1. La facette ctype

La facette ctype dérive d'une classe de base dans laquelle sont définies les différentes catégories de caractères. Cette classe est déclarée comme suit dans l'en-tête `locale` :

```
class ctype_base
{
public:
    enum mask
    {
        space = SPACE_VALUE, print = PRINT_VALUE,
        cntrl = CNTRL_VALUE, alpha = ALPHA_VALUE,
        digit = DIGIT_VALUE, xdigit = XDIGIT_VALUE,
```

```
        upper = UPPER_VALUE, lower = LOWER_VALUE,
        punct = PUNCT_VALUE,
        alnum = alpha | digit, graph = alnum | punct
    };
};
```

Les valeurs numériques utilisées par cette énumération sont définies de telle manière que les constantes de type mask constituent un champ de bits. Ainsi, il est possible de définir des combinaisons entre ces valeurs, certains caractères pouvant appartenir à plusieurs catégories en même temps. Deux combinaisons standards sont d'ailleurs définies, `alnum`, qui caractérise les caractères alphanumériques, et `graph`, qui représente tous les caractères alphanumériques et de ponctuation. Les autres constantes permettent de caractériser les caractères selon leur nature et leur signification est en général claire. La seule constante qui dont l'interprétation n'est pas immédiate est la constante `xdigit`, qui identifie tous les caractères pouvant servir de chiffre dans la notation des nombres hexadécimaux. Cela comprend les chiffres normaux et les lettres '`A`' à '`F`'.

La classe `template` ctype quant à elle est déclarée comme suit dans l'en-tête `locale` :

```
template <class charT>
class ctype : public locale::facet, public ctype_base
{
public:
// Les types de données :
    typedef charT char_type;

// Le constructeur :
    explicit ctype(size_t refs = 0);

// Les méthode de classification :
    bool        is(mask m, charT c) const;
    const charT *is(const charT *premier, const charT *dernier,
        mask *vecteur) const;
    const charT *scan_is(mask m,
        const charT *premier, const charT *dernier) const;
    const charT *scan_not(mask m,
        const charT *premier, const charT *dernier) const;
    charT        toupper(charT c) const;
    const charT *toupper(const charT *premier, const charT *dernier) const;
    charT        tolower(charT c) const;
    const charT *tolower(const charT *premier, const charT *dernier) const;
    charT        widen(char c) const;
    const charT *widen(const char *premier, const char *dernier,
        charT *destination) const;
    char         narrow(charT c, char defaut) const;
    const char  *narrow(const charT *premier, const charT *dernier,
        char defaut, char *destination) const;

// L'identificateur de la facette :
    static locale::id id;
};
```

Note : Comme pour toutes les facettes standards, les méthodes publiques délèguent leur travail à des méthodes virtuelles déclarées en zone protégée dont le nom est celui de la méthode publique préfixé par la chaîne de caractères « `do_` ». Ces méthodes peuvent être redéfinies par les classes dérivées de la facette ctype et font donc partie de l'interface des facettes standards. Cependant,

elles ne sont pas représentées dans la déclaration donnée ci-dessus par souci de simplicité. Leur sémantique est exactement la même que celle des méthodes publiques correspondantes. Nous verrons dans la Section 16.3.2 la manière de procéder pour redéfinir certaines des méthodes des facettes standards.

Les méthodes `scan_is` et `scan_not` permettent de rechercher un caractère selon un critère particulier dans un tableau de caractères. La méthode `scan_is` recherche le premier caractère qui est du type indiqué par son paramètre `m`, et la méthode `scan_not` le premier caractère qui n'est pas de ce type. Ces deux méthodes prennent en paramètre un pointeur sur le premier caractère du tableau dans lequel la recherche doit s'effectuer et le pointeur suivant l'emplacement du dernier caractère de ce tableau. Elles renvoient toutes les deux un pointeur référençant le caractère trouvé, ou le pointeur de fin si aucun caractère ne vérifie le critère spécifié.

Les autres méthodes de la facette ctype sont fournies sous deux versions. La première permet d'effectuer une opération sur un caractère unique et la deuxième permet de reproduire cette opération sur une séquence de caractères consécutifs. Dans ce dernier cas, les caractères sur lesquels l'opération peut être effectuée sont spécifiés à l'aide de deux pointeurs, l'un sur le premier caractère et l'autre sur le caractère suivant le dernier caractère de la séquence, comme il est d'usage de le faire dans tous les algorithmes de la bibliothèque standard.

Les deux méthodes `is` permettent donc respectivement de déterminer si un caractère est du type indiqué par le paramètre `m` ou non, ou d'obtenir la suite des descriptions de chaque caractère dans le tableau de valeur de type mask pointé par le paramètre `vecteur`. De même, les méthodes `toupper` et `tolower` permettent respectivement de convertir un caractère unique ou tous les caractères d'un tableau en majuscule ou en minuscule. La méthode `widen` permet de transtyper un caractère ou tous les caractères d'un tableau de type char en caractères du type par lequel la classe ctype est paramétrée. Enfin, les méthodes `narrow` permettent de réaliser l'opération inverse, ce qui peut provoquer une perte de données puisque le type char est le plus petit des types de caractères qui puisse exister. Il est donc possible que le transtypage ne puisse se faire, dans ce cas, les méthodes `narrow` utilisent la valeur par défaut spécifiée par le paramètre `defaut`.

Exemple 16-2. Conversion d'une wstring en string

```
#include <iostream>
#include <string>
#include <locale>

using namespace std;

int main(void)
{
    // Fixe la locale globale aux préférences de l'utilisateur :
    locale::global(locale(""));
    // Lit une chaîne de caractères larges :
    wstring S;
    wcin >> S;
    // Récupère la facette ctype<wchar_t> de la locale courante :
    const ctype<wchar_t> &f =
        use_facet<ctype<wchar_t> >(locale());
    // Construit un tampon pour recevoir le résultat de la conversion :
    size_t l = S.length() + 1;
    char *tampon = new char[l];
    // Effectue la conversion :
    f.narrow(S.c_str(), S.c_str() + l, 'E', tampon);
```

```
    // Affiche le résultat :
    cout << tampon << endl;
 delete[] tampon;
    return 0;
}
```

> **Note :** Les conversions effectuées par les méthodes `narrow` et `widen` ne travaillent qu'avec les représentations de caractères classiques du langage C++. Cela signifie que les caractères sont tous représentés par une unique valeur de type char ou wchar_t (ces méthodes n'utilisent donc pas de représentation des caractères basées sur des séquences de caractères de longueurs variables). La méthode `narrow` de l'exemple précédent écrit donc autant de caractères dans le tampon destination qu'il y en a dans la chaîne à convertir.

Vous constaterez que l'utilisation de la méthode `is` pour déterminer la nature des caractères peut être relativement fastidieuse, car il faut récupérer la facette ctype, déterminer la valeur du masque à utiliser, puis appeler la méthode. La bibliothèque standard définit donc un certain nombre de fonctions globales utilitaires dans l'en-tête `locale` :

```
template <class charT> bool isspace (charT c, const locale &l) const;
template <class charT> bool isprint (charT c, const locale &l) const;
template <class charT> bool iscntrl (charT c, const locale &l) const;
template <class charT> bool isupper (charT c, const locale &l) const;
template <class charT> bool islower (charT c, const locale &l) const;
template <class charT> bool isalpha (charT c, const locale &l) const;
template <class charT> bool isdigit (charT c, const locale &l) const;
template <class charT> bool ispunct (charT c, const locale &l) const;
template <class charT> bool isxdigit(charT c, const locale &l) const;
template <class charT> bool isalnum (charT c, const locale &l) const;
template <class charT> bool isgraph (charT c, const locale &l) const;
template <class charT> charT toupper(charT c, const locale &l) const;
template <class charT> charT tolower(charT c, const locale &l) const;
```

L'utilisation de ces fonctions ne doit pas poser de problème particulier. Elles prennent toutes en premier paramètre le caractère à caractériser et en deuxième paramètre la locale dont la facette ctype doit être utilisée pour réaliser cette caractérisation. Chaque fonction permet de tester le caractère pour l'appartenance à l'une des catégories de caractères définies dans la classe de base ctype_base. Notez cependant que si un grand nombre de caractères doivent être caractérisés pour une même locale, il est plus performant d'obtenir la facette ctype de cette locale une bonne fois pour toutes et d'effectuer les appels à la méthode `is` en conséquence.

La classe ctype étant une classe `template`, elle peut être utilisée pour n'importe quel type de caractère a priori. Toutefois, il est évident que cette classe peut être optimisée pour les types de caractère simples, et tout particulièrement pour le type char, parce qu'il ne peut pas prendre plus de 256 valeurs différentes. La bibliothèque standard définit donc une spécialisation totale de la classe `template` ctype pour le type char. L'implémentation de cette spécialisation se base sur un tableau de valeurs de type mask indexée par les valeurs que peuvent prendre les variables de type char. Ce tableau permet donc de déterminer rapidement les caractéristiques de chaque caractère existant. Le constructeur de cette spécialisation diffère légèrement du constructeur de sa classe `template` car il peut prendre en paramètre un pointeur sur ce tableau de valeurs et un booléen indiquant si ce tableau doit être détruit automatiquement par la facette lorsqu'elle est elle-même détruite ou non. Ce constructeur prend également en troisième paramètre une valeur de type entier indiquant, comme pour toutes les facettes standards, si la locale doit prendre en charge la gestion de la durée de vie de la facette ou non.

Les autres méthodes de cette spécialisation sont identiques aux méthodes de la classe `template` de base et ne seront donc pas décrites ici.

16.2.2.2. La facette codecvt

La facette codecvt permet de réaliser les opérations de conversion d'un mode de représentation des caractères à un autre. En général, en informatique, les caractères sont codés par des nombres. Le type de ces nombres, ainsi que la manière de les utiliser, peut varier grandement d'une représentation à une autre, et les conversions peuvent ne pas se faire simplement. Par exemple, certaines représentations codent chaque caractère avec une valeur unique du type de caractère utilisé, mais d'autres codent les caractères sur des séquences de longueur variable. On ne peut dans ce cas bien entendu pas convertir directement une représentation en une autre, car l'interprétation que l'on peut faire des nombres représentant les caractères dépend du contexte déterminé par les nombres déjà lus. Les opérations de conversion ne sont donc pas toujours directes.

De plus, dans certains encodages à taille variable, l'interprétation des caractères peut dépendre des caractères déjà convertis. La facette codecvt maintient donc un état pendant les conversions qu'elle effectue, état qui lui permet de reprendre la conversion d'une séquence de caractères dans le cas de conversions réalisées en plusieurs passes. Bien entendu, tous les encodages ne nécessitent pas forcément le maintien d'un tel état. Cependant, certains l'exigent et il faut donc toujours le prendre en compte dans les opérations de conversion si l'on souhaite que le programme soit portable. Pour les séquences de caractères à encodage variable utilisant le type de caractère de base char, le type de la variable d'état permettant de stocker l'état courant du convertisseur est le type mbstate_t. D'autres types peuvent être utilisés pour les séquences basées sur des types de caractères différents du type char, mais en général, tous les encodages à taille variable se basent sur ce type. Quoi qu'il en soit, la classe codecvt définit un type de donnée capable de stocker l'état d'une conversion partielle. Ce type est le type state_type, qui pourra donc toujours être récupéré dans la classe codecvt. La variable d'état du convertisseur devra être systématiquement fournie aux méthodes de conversion de la facette codecvt et devra bien entendu être initialisée à sa valeur par défaut au début de chaque nouvelle conversion.

> **Note :** La facette codecvt permet de réaliser les conversions d'une représentation des caractères à une autre, mais n'a pas pour but de changer l'encodage des caractères, c'est-à-dire l'association qui est faite entre les séquences de nombres et les caractères. Cela signifie que la facette codecvt permet par exemple de convertir des chaînes de caractères larges wchar_t en séquences de longueurs variables de caractères de type char, mais elle ne permet pas de passer d'une page de codes à une autre.

La facette codecvt dérive d'une classe de base nommée codecvt_base. Cette classe définit les différents résultats que peuvent avoir les opérations de conversion. Elle est déclarée comme suit dans l'en-tête `locale` :

```
class codecvt_base
{
public:
    enum result
    {
        ok, partial, error, noconv
    };
};
```

Comme vous pouvez le constater, une conversion peut se réaliser complètement (code de résultat `ok`), partiellement par manque de place dans la séquence destination ou par manque de données en entrées (code `partial`), ou pas du tout, soit en raison d'une erreur de conversion (code d'erreur `error`), soit parce qu'aucune conversion n'est nécessaire (code de résultat `noconv`).

La classe `template` codecvt elle-même est définie comme suit dans l'en-tête `locale` :

```
template <class internT, class externT, class stateT>
class codecvt : public locale::facet, public codecvt_base
{
public:
// Les types de données :
    typedef internT intern_type;
    typedef externT extern_type;
    typedef stateT  state_type;

// Le constructeur :
    explicit codecvt(size_t refs=0);

// Les fonctions de conversion :
    result out(stateT &etat, const internT *premier,
        const internT *dernier, const internT *&suiv_source,
        externT *dernier, externT *limite, externT *&suiv_dest) const;
    result in(stateT &etat, const externT *premier,
        const externT *dernier, const externT *&suiv_source,
        internT *dernier, internT *limite, internT *&suiv_dest) const;
    result unshift(stateT &etat,
        externT *dernier, externT *limite, externT *&suiv_dest) const;
    int length(const stateT &etat,
        const externT *premier, const externT *dernier, size_t max) const;
    int max_length() const throw();
    int encoding() const throw();
    bool always_noconv() const throw();

// L'identificateur de la facette :
    static locale::id id;
};
```

> **Note :** Les méthodes virtuelles d'implémentation des méthodes publiques n'ont pas été écrites dans la déclaration précédente par souci de simplification. Elles existent malgré tout, et peuvent être redéfinies par les classes dérivées afin de personnaliser le comportement de la facette.

Cette classe `template` est paramétrée par le type de caractère interne à la classe codecvt, par un deuxième type de caractère qui sera par la suite dénommé type externe, et par le type des variables destinées à recevoir l'état courant d'une conversion. Les implémentations de la bibliothèque standard doivent obligatoirement instancier cette classe `template` pour les types char et wchar_t. Le type de gestion de l'état des conversions utilisé est alors le type prédéfini mbstate_t, qui permet de conserver l'état des conversions entre le type natif wchar_t et les séquences de caractères simples à taille variable. Ainsi, vous pourrez toujours utiliser les instances codecvt<wchar_t, char, mbstate_t> et codecvt<char, char, mbstate_t> de la facette codecvt dans vos programmes. Si vous désirez réaliser des conversions pour d'autres types de caractères, vous devrez fournir vous-même des spécialisations de la facette codecvt.

Les méthodes `in` et `out` permettent respectivement, comme leurs signatures l'indiquent, de réaliser les conversions entre les types interne et externe et vice versa. Elles prennent toutes deux sept paramètres. Le premier paramètre est une référence sur la variable d'état qui devra être fournie à chaque appel lors de conversions successives d'un même flux de données. Cette variable est destinée à recevoir l'état courant de la conversion et permettra aux appels suivants de convertir correctement les caractères suivants du flux d'entrée. Les deux paramètres suivants permettent de spécifier la séquence de caractères à convertir. Ils doivent contenir le pointeur sur le début de la séquence et le pointeur sur le caractère suivant le dernier caractère de la séquence. Le quatrième paramètre est un paramètre de retour, la fonction lui affectera la valeur du pointeur où la conversion s'est arrêtée. Une conversion peut s'arrêter à cause d'une erreur ou tout simplement parce que le tampon destination ne contient pas assez de place pour accueillir un caractère de plus. Ce pointeur pourra être utilisé dans un appel ultérieur comme pointeur de départ avec la valeur de la variable d'état à l'issue de la conversion pour effectuer la suite de cette conversion. Enfin, les trois derniers paramètres spécifient le tampon destination dans lequel la séquence convertie doit être écrite. Ils permettent d'indiquer le pointeur de début de ce tampon, le pointeur suivant le dernier emplacement utilisable, et un pointeur de retour qui indiquera la dernière position écrite par l'opération de conversion. Ces deux méthodes renvoient une des constantes de l'énumération result définie dans la classe de base codecvt_base pour indiquer comment la conversion s'est effectuée. Si aucune conversion n'est nécessaire, les pointeurs sur les caractères suivants sont initialisés à la valeur des pointeurs de début de séquence et aucune écriture n'a lieu dans le tampon destination.

Exemple 16-3. Conversion d'une chaîne de caractères larges en chaîne à encodage variable

```
#include <iostream>
#include <string>
#include <locale>

using namespace std;

int main(void)
{
    // Fixe la locale globale :
    locale::global(locale(""));
    // Lit une ligne :
    wstring S;
    getline(wcin, S);
    // Récupère la facette de conversion vers wchar_t :
    const codecvt<wchar_t, char, mbstate_t> &f =
        use_facet<codecvt<wchar_t, char, mbstate_t> >(locale());
    // Effectue la conversion :
    const wchar_t *premier = S.c_str();
    const wchar_t *dernier = premier + S.length();
    const wchar_t *suivant = premier;
    string s;
    char tampon[10];
    char *fincvt = tampon;
    codecvt_base::result r;
    mbstate_t etat = mbstate_t();
    while (premier != dernier)
    {
        // Convertit un morceau de la chaîne :
        r = f.out(etat, premier, dernier, suivant,
            tampon, tampon+10, fincvt);
        // Vérifie les erreurs possibles :
        if (r == codecvt_base::ok || r == codecvt_base::partial)
```

```
            cout << "." << flush;
        else if (r == codecvt_base::noconv)
        {
            cout << "conversion non nécessaire" << endl;
            break;
        }
        else if (r == codecvt_base::error)
        {
            cout << "erreur" << endl;
            cout << suivant - premier << endl;
            cout << fincvt - tampon << endl;
            break ;
        }
        // Récupère le résultat et prépare la conversion suivante :
        s.append(tampon, fincvt - tampon);
        premier = suivant;
    }
    cout << endl;
    // Affiche le résultat :
    cout << s << endl;
    return 0;
}
```

Note : Si l'on désire effectuer une simple conversion d'une chaîne de caractères de type wchar_t en chaîne de caractères C classique, on cherchera plutôt à utiliser la méthode `narrow` de la facette ctype présentée dans la section précédente. En effet, la facette codecvt utilise, a priori, une séquence de caractères avec un encodage à taille variable, ce qui ne correspond pas à la représentation des chaînes de caractères C classiques, pour lesquelles chaque valeur de type char représente un caractère.

Il est possible de compléter une séquence de caractères à encodage variable de telle sorte que la variable d'état du convertisseur soit réinitialisée. Cela permet de terminer une chaîne de caractères partiellement convertie, ce qui en pratique revient à compléter la séquence de caractères avec les données qui représenteront le caractère nul terminal. Cette opération peut être réalisée à l'aide de la méthode `unshift` de la facette codecvt. Cette méthode prend en paramètre une référence sur la variable d'état du convertisseur, ainsi que les pointeurs de début et de fin du tampon dans lequel les valeurs à ajouter sont écrites. Le dernier paramètre de la méthode `unshift` est une référence sur un pointeur qui recevra l'adresse suivant celle la dernière valeur écrite par la méthode si l'opération se déroule correctement.

Il va de soi que la détermination de la longueur d'une chaîne de caractères dont les caractères ont une représentation à taille variable n'est pas simple. La facette codecvt comporte donc une méthode `length` permettant de calculer, en nombre de caractères de type intern_type, la longueur d'une séquence de caractères de type extern_type. Cette méthode prend en paramètre la variable d'état du convertisseur ainsi que les pointeurs spécifiant la séquence de caractères dont la longueur doit être calculée. Le dernier paramètre est la valeur maximale que la fonction peut retourner. Elle permet de limiter la détermination de la longueur de la séquence source à une borne maximale, par exemple la taille d'un tampon destination. La valeur retournée est bien entendu la longueur de cette séquence ou, autrement dit, le nombre de valeurs de type intern_type nécessaires pour stocker le résultat de la conversion que la méthode `in` ferait avec les mêmes paramètres. D'autre part, il est possible de déterminer le nombre maximal de valeurs de type intern_type nécessaires pour représenter un unique caractère représenté par une séquence de caractères de type extern_type. Pour cela, il suffit d'appeler la méthode `max_length` de la facette codecvt.

Exemple 16-4. Détermination de la longueur d'une chaîne de caractères à encodage variable

```
#include <iostream>
#include <string>
#include <locale>
#include <limits>

using namespace std;

int main(void)
{
    // Fixe la locale globale :
    locale::global(locale(""));
    // Lit une ligne :
    string s;
    getline(cin, s);
    // Récupère la facette de conversion vers wchar_t :
    const codecvt<wchar_t, char, mbstate_t> &f =
        use_facet<codecvt<wchar_t, char, mbstate_t> >(locale());
    // Affiche la longueur de la chaîne d'entrée :
    int l1 = s.length();
    // Calcule la longueur de la ligne en wchar_t :
    mbstate_t etat = mbstate_t();
    int l2 = f.length(etat, s.c_str(), s.c_str() + l1,
        numeric_limits<size_t>::max());
    // Affiche les deux longueurs :
    cout << l1 << endl;
    cout << l2 << endl;
    return 0;
}
```

Comme on l'a déjà indiqué ci-dessus, toutes les représentations des caractères ne sont pas à taille variable et toutes les représentations ne nécessitent pas forcément l'utilisation d'une variable d'état de type state_type. Vous pouvez déterminer dynamiquement si le mode de représentation des caractères du type intern_type utilise un encodage à taille variable ou non à l'aide de la méthode `encoding`. Cette méthode renvoie `-1` si la représentation des caractères de type extern_type dépend de l'état du convertisseur, ou le nombre de caractères de type extern_type nécessaires au codage d'un caractère de type intern_type si ce nombre est constant. Si la valeur renvoyée est `0`, ce nombre n'est pas constant, mais, contrairement à ce qui se passe lorsque la valeur renvoyée est `-1`, ce nombre ne dépend pas de la valeur de la variable d'état du convertisseur.

Enfin, certains modes de représentation des caractères sont compatibles, voire franchement identiques. Dans ce cas, jamais aucune conversion n'est réalisée, et les méthodes `in` et `out` renvoient toujours `noconv`. C'est par exemple le cas de la spécialisation codecvt<char, char, mbstate_t> de la facette codecvt. Vous pouvez déterminer si une facette effectuera des conversions ou non en appelant la méthode `always_noconv`. Elle retourne `true` si jamais aucune conversion ne se fera et `false` sinon.

16.2.3. Les facettes de comparaison de chaînes

Les chaînes de caractères sont généralement classées par ordre alphabétique, ou, plus précisément, dans l'ordre lexicographique. L'ordre lexicographique est l'ordre défini par la séquence des symboles lexicaux utilisés (c'est-à-dire les symboles utilisés pour former les mots du langage, donc, en pratique, les lettres, les nombres, la ponctuation, etc.). Cet ordre est celui qui est défini par la comparaison successive des caractères des deux chaînes à comparer, le premier couple de caractères différents

permettant de donner un jugement de classement. Ainsi, les chaînes les plus petites au sens de l'ordre lexicographique sont les chaînes qui commencent par les premiers symboles du lexique utilisé. Cette manière de procéder suppose bien entendu que les symboles utilisés pour former les mots du lexique sont classés dans un ordre correct. Par exemple, il faut que la lettre 'a' apparaisse avant la lettre 'b', qui elle-même doit apparaître avant la lettre 'c', etc.

Malheureusement, cela n'est pas si simple, car cet ordre n'est généralement pas celui utilisé par les pages de codes d'une part, et il existe toujours des symboles spéciaux dont la classification nécessite un traitement spécial d'autre part. Par exemple, les caractères accentués sont généralement placés en fin de page de code et apparaissent donc à la fin de l'ordre lexicographique, ce qui perturbe automatiquement le classement des chaînes de caractères contenant des accents. De même, certaines lettres sont en réalité des compositions de lettres et doivent être prises en compte en tant que telles dans les opérations de classement. Par exemple, la lettre 'æ' doit être interprétée comme un 'a' suivi d'un 'e'. Et que dire du cas particulier des majuscules et des minuscules ?

Comme vous pouvez le constater, il n'est pas possible de se baser uniquement sur l'ordre des caractères dans leur page de code pour effectuer les opérations de classement de chaînes de caractères. De plus, il va de soi que l'ordre utilisé pour classer les symboles lexicographiques dépend de ces symboles et donc de la locale utilisé. La bibliothèque standard fournit donc une facette prenant en compte tous ces paramètres : la classe `template` collate.

Le principe de fonctionnement de la facette collate est de transformer les chaînes de caractères utilisant les conventions de la locale à laquelle la facette appartient en une chaîne de caractères indépendante de la locale, comprenant éventuellement des codes de contrôle spéciaux pour les caractères spécifiques à cette locale. Les chaînes de caractères ainsi transformées peuvent alors être comparées entre elles directement, avec les méthodes de comparaison classique de chaînes de caractères qui utilisent l'ordre lexicographique du jeu de caractères du langage C. La transformation est effectuée de telle manière que cette comparaison produit le même résultat que la comparaison tenant compte de la locale des chaînes de caractères non transformées.

La facette collate est déclarée comme suit dans l'en-tête `locale` :

```
template <class charT>
class collate : public locale::facet
{
public:
// Les types de données :
    typedef charT               char_type;
    typedef basic_string<charT> string_type;

// Le constructeur :
    explicit collate(size_t refs = 0);

// Les méthodes de comparaison de chaînes :
    string_type transform(const charT *debut, const charT *fin) const;
    int compare(const charT *deb_premier, const charT *fin_premier,
        const charT *deb_deuxieme, const charT *fin_deuxieme) const;
    long hash(const charT *debut, const charT *fin) const;

// L'identificateur de la facette :
    static locale::id id;
};
```

> **Note :** Les méthodes virtuelles d'implémentation des méthodes publiques n'ont pas été écrites dans la déclaration précédente par souci de simplification. Elles existent malgré tout, et peuvent être redéfinies par les classes dérivées afin de personnaliser le comportement de la facette.

La méthode `transform` est la méthode fondamentale de la facette collate. C'est cette méthode qui permet d'obtenir la chaîne de caractères transformée. Elle prend en paramètre le pointeur sur le début de la chaîne de caractères à transformer et le pointeur sur le caractère suivant le dernier caractère de cette chaîne. Elle retourne une basic_string contenant la chaîne transformée, sur laquelle les opérations de comparaison classiques pourront être appliquées.

Il est possible d'effectuer directement la comparaison entre deux chaînes de caractères, sans avoir à récupérer les chaînes de caractères transformées. Cela peut être réalisé grâce à la méthode `compare`, qui prend en paramètre les pointeurs de début et de fin des deux chaînes de caractères à comparer et qui renvoie un entier indiquant le résultat de la comparaison. Cet entier est négatif si la première chaîne est inférieure à la deuxième, positif si elle est supérieure, et nul si les deux chaînes sont équivalentes.

Exemple 16-5. Comparaison de chaînes de caractères localisées

```
#include <iostream>
#include <string>
#include <locale>

using namespace std;

int main(void)
{
    // Fixe la locale globale :
    locale::global(locale(""));
    // Lit deux lignes en entrée :
    cout << "Entrez la première ligne :" << endl;
    string s1;
    getline(cin, s1);
    cout << "Entrez la deuxième ligne :" << endl;
    string s2;
    getline(cin, s2);
    // Récupère la facette de comparaison de chaînes :
    const collate<char> &f =
        use_facet<collate<char> >(locale());
    // Compare les deux chaînes :
    int res = f.compare(
        s1.c_str(), s1.c_str() + s1.length(),
        s2.c_str(), s2.c_str() + s2.length());
    if (res < 0)
    {
        cout << "\"" << s1 << "\" est avant \"" <<
            s2 << "\"." << endl;
    }
    else if (res > 0)
    {
        cout << "\"" << s1 << "\" est après \"" <<
            s2 << "\"." << endl;
    }
    else
    {
        cout << "\"" << s1 << "\" est égale à \"" <<
```

```
            s2 << "\"." << endl;
    }    return 0;
}
```

> **Note :** La méthode `compare` est très pratique pour comparer deux chaînes de caractères de manière ponctuelle. Cependant, on lui préférera la méthode `transform` si un grand nombre de comparaisons doit être effectué. En effet, il est plus simple de transformer toutes les chaînes de caractères une bonne fois pour toutes et de travailler ensuite directement sur les chaînes transformées. Ce n'est que lorsque les opérations de comparaison auront été terminées que l'on pourra revenir sur les chaînes de caractères initiales. On évite ainsi de faire des transformation à répétition des chaînes à comparer et on gagne ainsi beaucoup de temps. Bien entendu, cela nécessite de conserver l'association entre les chaînes de caractères transformées et les chaînes de caractères initiales, et donc de doubler la consommation mémoire du programme due au chaînes de caractères pendant le traitement de ces chaînes.

Enfin, il est courant de chercher à déterminer une clef pour chaque chaîne de caractères. Cette clef peut être utilisée pour effectuer une recherche rapide des chaînes de caractères. La méthode `hash` de la facette collate permet de calculer une telle clef, en garantissant que deux chaînes de caractères identiques au sens de la méthode `compare` auront la même valeur de clef. On notera cependant que cette clef n'est pas unique, deux chaînes de caractères peuvent avoir deux valeurs de clefs identiques même si la méthode `compare` renvoie une valeur non nulle. Cependant, ce cas est extrêmement rare, et permet d'utiliser malgré tout des algorithmes de recherche rapide. La seule chose à laquelle il faut faire attention est que ces algorithmes doivent pouvoir supporter les clefs multiples.

> **Note :** Les clefs à probabilité de recouvrement faible comme celle retournée par la méhtode `hash` sont généralement utilisées dans les structures de données appelées *tables de hachage*, ce qui explique le nom donné à cette méthode. Les tables de hachage sont en réalité des tableaux de listes chaînées indexés par la clef de hachage (si la valeur de la clef dépasse la taille du tableau, elle est ramenée dans les limites de celui-ci par une opération de réduction). Ce sont des structures permettant de rechercher rapidement des valeurs pour lesquelles une fonction de hachage simple existe. Cependant, elles se comportent moins bien que les arbres binaires lorsque le nombre d'éléments augmente (quelques milliers). On leur préférera donc généralement les associations de la bibliothèque standard, comme les map et multimap par exemple. Vous pouvez consulter la bibliographie si vous désirez obtenir plus de renseignements sur les tables de hachage et les structures de données en général. Les associations et les conteneurs de la bibliothèque standard seront décrites dans le Chapitre 17.

16.2.4. Les facettes de gestion des nombres

Les opérations de formatage et les opérations d'interprétation des données numériques dépendent bien entendu des conventions nationales de la locale incluse dans les flux qui effectuent ces opérations. En réalité, ces opérations ne sont pas prises en charge directement par les flux, mais plutôt par les facettes de gestion des nombres, qui regroupent toutes les opérations propres aux conventions nationales.

La bibliothèque standard définit en tout trois facettes qui interviennent dans les opérations de formatage : une facette utilitaire, qui contient les paramètres spécifiques à la locale, et deux facettes dédiées respectivement aux opérations de lecture et aux opérations d'écriture des nombres.

16.2.4.1. La facette num_punct

La facette qui regroupe tous les paramètres de la locale est la facette num_punct. Elle est déclarée comme suit dans l'en-tête `locale` :

```
template <class charT>
class numpunct : public locale::facet
{
public:
// Les types de données :
    typedef charT               char_type;
    typedef basic_string<charT> string_type;

// Le constructeur :
    explicit numpunct(size_t refs = 0);

// Les méthodes de lecture des options de formatage des nombres :
    char_type    decimal_point()   const;
    char_type    thousands_sep()   const;
    string       grouping()        const;
    string_type  truename()        const;
    string_type  falsename()       const;

// L'identificateur de la facette :
    static locale::id id;
};
```

> **Note :** Les méthodes virtuelles d'implémentation des méthodes publiques n'ont pas été écrites dans la déclaration précédente par souci de simplification. Elles existent malgré tout, et peuvent être redéfinies par les classes dérivées afin de personnaliser le comportement de la facette.

La méthode `decimal_point` permet d'obtenir le caractère qui doit être utilisé pour séparer le chiffre des unités des chiffres après la virgule lors des opérations de formatage des nombres à virgule. La valeur par défaut est le caractère '`.`', mais en France, le caractère utilisé est la virgule (caractère '`,`'). De même, la méthode `thousands_sep` permet de déterminer le caractère qui est utilisé pour séparer les groupes de chiffres lors de l'écriture des grands nombres. La valeur par défaut renvoyée par cette fonction est le caractère virgule (caractère '`,`'), mais dans les locales françaises, on utilise généralement un espace (caractère ' '). Enfin, la méthode `grouping` permet de déterminer les emplacements où ces séparateurs doivent être introduits. La chaîne de caractères renvoyée détermine le nombre de chiffres de chaque groupe de chiffres. Le nombre de chiffres du premier groupe est ainsi stocké dans le premier caractère de la chaîne de caractères renvoyée par la méthode `grouping`, celui du deuxième groupe est stocké dans le deuxième caractère, et ainsi de suite. Le dernier nombre ainsi obtenu dans cette chaîne de caractères est ensuite utilisé pour tous les groupes de chiffres suivants, ce qui évite d'avoir à définir une chaîne de caractères arbitrairement longue. Un nombre de chiffres nul indique que le mécanisme de groupage des chiffres des grands nombres est désactivé. Les facettes de la plupart des locales renvoient la valeur `"\03"`, ce qui permet de grouper les chiffres par paquets de trois (milliers, millions, milliards, etc.).

> **Note :** Remarquez que les valeurs stockées dans la chaîne de caractères renvoyée par la méthode `grouping` sont des valeurs numériques et non des chiffres formatés dans la chaîne de caractères. Ainsi, la valeur par défaut renvoyée est bien `"\03"` et non `"3"`.

Les méthodes `truename` et `falsename` quant à elles permettent aux facettes de formatage d'obtenir les chaînes de caractères qui représentent les valeurs `true` et `false` des booléens. Ce sont ces chaînes de caractères qui sont utilisées lorsque l'option de formatage `boolalpha` a été activée dans les flux d'entrée / sortie. Les valeurs retournées par ces méthodes sont, par défaut, les mots anglais `true` et `false`. Il est concevable dans d'autres locales, cependant, d'avoir des noms différents pour ces deux valeurs. Nous verrons dans la Section 16.3.2 la manière de procéder pour redéfinir ces méthodes et construire ainsi une locale personnalisée et francisée.

> **Note :** Bien entendu, les facettes d'écriture et de lecture des nombres utilisent également les options de formatage qui sont définis au niveau des flux d'entrée / sortie. Pour cela, les opérations d'entrée / sortie reçoivent en paramètre une référence sur le flux contenant ces options.

16.2.4.2. La facette d'écriture des nombres

L'écriture et le formatage des nombres sont pris en charge par la facette num_put. Cette facette est déclarée comme suit dans l'en-tête `locale` :

```
template <class charT,
    class OutputIterator = ostreambuf_iterator<charT> >
class num_put : public locale::facet
{
public:
// Les types de données :
    typedef charT          char_type;
    typedef OutputIterator iter_type;

// Le constructeur :
    explicit num_put(size_t refs = 0);

// Les méthodes d'écriture des nombres :
    iter_type put(iter_type s, ios_base &f, char_type remplissage, bool v) const;
    iter_type put(iter_type s, ios_base &f, char_type remplissage, long v) const;
    iter_type put(iter_type s, ios_base &f, char_type remplissage,
        unsigned long v) const;
    iter_type put(iter_type s, ios_base &f, char_type remplissage,
        double v) const;
    iter_type put(iter_type s, ios_base &f, char_type remplissage,
        long double v) const;
    iter_type put(iter_type s, ios_base &f, char_type remplissage,
        void *v) const;

// L'identificateur de la facette :
    static locale::id id;
};
```

> **Note :** Les méthodes virtuelles d'implémentation des méthodes publiques n'ont pas été écrites dans la déclaration précédente par souci de simplification. Elles existent malgré tout, et peuvent être redéfinies par les classes dérivées afin de personnaliser le comportement de la facette.

Comme vous pouvez le constater, cette facette dispose d'une surcharge de la méthode `put` pour chacun des types de base du langage. Ces surcharges prennent en paramètre un itérateur d'écriture sur le flux de sortie sur lequel les données formatées devront être écrites, une référence sur le flux de sortie contenant les options de formatage à utiliser lors du formatage des nombres, le caractère de remplissage à utiliser, et bien entendu la valeur à écrire.

En général, ces méthodes sont appelées au sein des opérateurs d'insertion `operator<<` pour chaque type de donnée existant. De plus, le flux de sortie sur lequel les écritures doivent être effectuées est le même que le flux servant à spécifier les options de formatage, si bien que l'appel aux méthodes `put` est extrêmement simplifié. Nous verrons plus en détail la manière d'appeler ces méthodes dans la Section 16.3.1, lorsque nous écrirons une nouvelle facette pour un nouveau type de donnée.

16.2.4.3. La facette de lecture des nombres

Les opérations de lecture des nombres à partir d'un flux de données sont prises en charge par la facette num_get. Cette facette est déclarée comme suit dans l'en-tête `locale` :

```
template <class charT,
    class InputIterator = istreambuf_iterator<charT> >
class num_get : public locale::facet
{
public:
// Les types de données :
    typedef charT         char_type;
    typedef InputIterator iter_type;

// Le constructeur :
    explicit num_get(size_t refs = 0);

// Les méthodes de lecture des nombres :
    iter_type get(iter_type in, iter_type end, ios_base &,
        ios_base::iostate &err, bool &v) const;
    iter_type get(iter_type in, iter_type end, ios_base &,
        ios_base::iostate &err, long &v) const;
    iter_type get(iter_type in, iter_type end, ios_base &,
        ios_base::iostate &err, unsigned short &v) const;
    iter_type get(iter_type in, iter_type end, ios_base &,
        ios_base::iostate &err, unsigned int &v) const;
    iter_type get(iter_type in, iter_type end, ios_base &,
        ios_base::iostate &err, unsigned long &v) const;
    iter_type get(iter_type in, iter_type end, ios_base &,
        ios_base::iostate &err, float &v) const;
    iter_type get(iter_type in, iter_type end, ios_base &,
        ios_base::iostate &err, double &v) const;
    iter_type get(iter_type in, iter_type end, ios_base &,
        ios_base::iostate &err, long double &v) const;
    iter_type get(iter_type in, iter_type end, ios_base &,
        ios_base::iostate &err, void *&v) const;

// L'identificateur de la facette :
    static locale::id id;
};
```

> **Note :** Les méthodes virtuelles d'implémentation des méthodes publiques n'ont pas été écrites dans la déclaration précédente par souci de simplification. Elles existent malgré tout, et peuvent être redéfinies par les classes dérivées afin de personnaliser le comportement de la facette.

Comme vous pouvez le constater, cette facette ressemble beaucoup à la facette num_put. Il existe en effet une surcharge de la méthode `get` pour chaque type de base du langage. Ces méthodes sont capables d'effectuer la lecture des données de ces types à partir du flux d'entrée, en tenant compte des paramètres des locales et des options de formatage du flux. Ces méthodes prennent en paramètre un itérateur de flux d'entrée, une valeur limite de cet itérateur au-delà de laquelle la lecture du flux ne se fera pas, la référence sur le flux d'entrée contenant les options de formatage et deux paramètres de retour. Le premier paramètre recevra un code d'erreur de type iostate qui pourra être positionné dans le flux d'entrée pour signaler l'erreur. Le deuxième est une référence sur la variable devant accueillir la valeur lue. Si une erreur se produit, cette variable n'est pas modifiée.

Les méthodes `get` sont généralement utilisées par les opérateurs d'extraction `operator>>` des flux d'entrée / sortie pour les types de données de base du langage. En général, ces opérateurs récupèrent la locale incluse dans le flux d'entrée sur lequel ils travaillent et utilisent la facette num_get de cette locale. Ils appellent alors la méthode `get` permettant de lire la donnée qu'ils doivent extraire, en fournissant ce même flux en paramètre. Ils testent ensuite la variable d'état retournée par la méthode `get` et, si une erreur s'est produite, modifient l'état du flux d'entrée en conséquence. Cette dernière opération peut bien entendu provoquer le lancement d'une exception, selon le masque d'exceptions utilisé pour le flux.

16.2.5. Les facettes de gestion des monnaies

La bibliothèque standard ne définit pas de type de donnée dédiés à la représentation des montants. Elle suppose en effet que les montants sont stockés dans des nombres à virgule flottante dans la plupart des programmes ou, pour les programmes qui désirent s'affranchir des erreurs d'arrondis inévitables lors de l'utilisation de flottants, sous forme textuelle dans des chaînes de caractères. En revanche, la bibliothèque standard fournit, tout comme pour les types standards, deux facettes localisées prenant en compte la lecture et l'écriture des montants. Ces facettes se basent également sur une troisième facette qui regroupe tous les paramètres spécifiques aux conventions nationales.

> **Note :** En réalité, les seuls types capables de représenter correctement les montants en informatique sont les entiers et les nombres à virgule fixe codés sur les entiers. En effet, les types intégraux sont les seuls types qui ne soulèvent pas de problème de représentation des nombres (à condition qu'il n'y ait pas de débordements bien entendu) et les nombres à virgule fixe sont particulièrement adaptés aux montants, car en général le nombre de chiffres significatifs après la virgule est fixé pour une monnaie donnée. Les nombres à virgule flottante ne permettent pas de représenter des valeurs avec précision et introduisent des erreurs incontrôlables dans les calculs et dans les arrondis. Les chaînes de caractères quant à elles souffrent de leur lourdeur et, dans la plupart des cas, de la nécessité de passer par des nombres à virgule flottantes pour interpréter leur valeur. Les facettes présentées dans cette section sont donc d'une utilité réduite pour les programmes qui cherchent à obtenir des résultats rigoureux et précis, et qui ne tolèrent pas les erreurs de représentation et les erreurs d'arrondis.

Toutes les facettes de gestion des montants sont des classes `template`. Cependant, contrairement aux autres facettes, ces facettes disposent d'un autre paramètre `template` que le type de caractère sur lequel elles travaillent. Ce paramètre est un paramètre de type booléen qui permet, selon sa valeur,

de spécifier si les facettes doivent travailler avec la représentation internationale des montants ou non. Il existe en effet une représentation universelle des montants qui, entre autres particularités, utilise les codes internationaux de monnaie (« USD » pour le dollar américain, « CAN » pour le dollar canadien, « EUR » pour l'euro, etc.).

Comme pour les facettes de gestion des nombres, les facettes prenant en charge les monnaies sont au nombre de trois. Une de ces trois facettes permet d'obtenir des informations sur la monnaie de la locale et les deux autres réalisent respectivement les opérations d'écriture et de lecture sur un flux.

16.2.5.1. La facette money_punct

La facette moneypunct est la facette permettant aux deux facettes d'écriture et de lecture des montants d'obtenir les informations relatives à la monnaie de leur locale. Cette facette est déclarée comme suit dans l'en-tête `locale` :

```
template <class charT, bool International = false>
class moneypunct : public locale::facet, public money_base
{
public:
// Les types de données :
    typedef charT char_type;
    typedef basic_string<charT> string_type;

// Le constructeur :
    explicit moneypunct(size_t refs = 0);

// Les méthodes de lecture des options de formatage des montants :
    charT       decimal_point() const;
    charT       thousands_sep() const;
    string      grouping()      const;
    int         frac_digits()   const;
    string_type curr_symbol()   const;
    pattern     pos_format()    const;
    pattern     neg_format()    const;
    string_type positive_sign() const;
    string_type negative_sign() const;
    static const bool intl = International;

// L'identificateur de la facette :
    static locale::id id;
};
```

> **Note :** Les méthodes virtuelles d'implémentation des méthodes publiques n'ont pas été écrites dans la déclaration précédente par souci de simplification. Elles existent malgré tout, et peuvent être redéfinies par les classes dérivées afin de personnaliser le comportement de la facette.

Comme vous pouvez le constater, cette facette dispose de méthodes permettant de récupérer les divers symboles qui sont utilisés pour écrire les montants de la monnaie qu'elle décrit. Ainsi, la méthode `decimal_point` renvoie le caractère qui doit être utilisé en tant que séparateur du chiffre des unités de la partie fractionnaire des montants, si celle-ci doit être représentée. De même, la méthode `thousands_sep` renvoie le caractère qui doit être utilisé pour séparer les groupes de chiffres pour

les grands montants, et la méthode `grouping` renvoie une chaîne contenant, dans chacun de ses caractères, le nombre de chiffres de chaque groupe. Ces méthodes sont donc semblables aux méthodes correspondantes de la facette numpunct. Le nombre de chiffres significatifs après la virgule utilisé pour cette monnaie peut être obtenue grâce à la méthode `frac_digits`. Ce n'est que si la valeur renvoyée par cette méthode est supérieure à 0 que le symbole de séparation des unités de la partie fractionnaire de la méthode `decimal_point` est utilisé.

La méthode `curr_symbol` permet d'obtenir le symbole monétaire de la monnaie. Ce symbole dépend de la valeur du paramètre `template International`. Si ce paramètre vaut `true`, le symbole monétaire renvoyé sera le symbole monétaire international. Dans le cas contraire, ce sera le symbole monétaire en usage dans le pays de circulation de la monnaie. La valeur du paramètre `International` pourra être obtenu grâce à la constante statique `intl` de la facette.

Les méthodes suivantes permettent de spécifier le format d'écriture des montants positifs et négatifs. Ces méthodes utilisent les définitions de constantes et de types de la classe de base money_base dont la facette moneypunct hérite. La classe money_base est déclarée comme suit dans l'en-tête `locale` :

```
class money_base
{
public:
    enum part
    {
        none, space, symbol, sign, value
    };
    struct pattern
    {
        char field[4];
    };
};
```

Cette classe contient la définition d'une énumération dont les valeurs permettent d'identifier les différentes composantes d'un montant, ainsi qu'une structure pattern qui contient un tableau de quatre caractères. Chacun de ces caractères peut prendre l'une des valeurs de l'énumération part. La structure pattern définit donc l'ordre dans lequel les composantes d'un montant doivent apparaître. Ce sont des motifs de ce genre qui sont renvoyés par les méthodes `pos_format` et `neg_format`, qui permettent d'obtenir respectivement le format des montants positifs et celui des montants négatifs.

Les différentes valeurs que peuvent prendre les éléments du motif pattern représentent chacune une partie de l'expression d'un montant. La valeur `value` représente bien entendu la valeur de ce montant, `sign` son signe et `symbol` le symbole monétaire. La valeur `space` permet d'insérer un espace dans l'expression d'un montant, mais les espaces ne peuvent pas être utilisés en début et en fin de montants. Enfin, la valeur `none` permet de ne rien mettre à la position où il apparaît dans le motif.

La manière d'écrire les montants positifs et négatifs varie grandement selon les pays. En général, il est courant d'utiliser le signe '`-`' pour signaler un montant négatif et aucun signe distinctif pour les montants positifs. Cependant, certains pays écrivent les montants négatifs entre parenthèses et la marque des montants négatifs n'est donc plus un simple caractère. Les méthodes `positive_sign` et `negative_sign` permettent d'obtenir les symboles à utiliser pour noter les montants positifs et négatifs. Elles retournent toutes les deux une chaîne de caractères, dont le premier est placé systématiquement à l'emplacement auquel la valeur `sign` a été affectée dans la chaîne de format renvoyée par les méthodes `pos_format` et `neg_format`. Les caractères résiduels, s'ils existent, sont placés à la fin de l'expression du montant complètement formatée. Ainsi, dans les locales pour lesquelles les montants négatifs sont écrits entre parenthèses, la chaîne renvoyée par la méthode `negative_sign`

est « () », et pour les locales utilisant simplement le signe négatif, cette chaîne ne contient que le caractère '-'.

16.2.5.2. Les facettes de lecture et d'écriture des montants

Les facettes d'écriture et de lecture des montants sont sans doute les facettes standards les plus simples. En effet, elles ne disposent que de méthodes permettant d'écrire et de lire les montants sur les flux. Ces facettes sont respectivement les facettes money_put et money_get. Elles sont définies comme suit dans l'en-tête `locale` :

```
template <class charT, bool Intl = false,
    class OutputIterator = ostreambuf_iterator<charT> >
class money_put : public locale::facet
{
public:
// Les types de données :
    typedef charT               char_type;
    typedef OutputIterator      iter_type;
    typedef basic_string<charT> string_type;

// Le constructeur :
    explicit money_put(size_t refs = 0);

// Les méthodes d'écriture des montants :
    iter_type put(iter_type s, bool intl, ios_base &f,
        char_type remplissage, long double units) const;
    iter_type put(iter_type s, bool intl, ios_base &f,
        char_type remplissage, const string_type &digits) const;

// L'identificateur de la facette :
    static locale::id id;
};

template <class charT,
    class InputIterator = istreambuf_iterator<charT> >
class money_get : public locale::facet
{
public:
// Les types de données :
    typedef charT               char_type;
    typedef InputIterator       iter_type;
    typedef basic_string<charT> string_type;

// Le constructeur :
    explicit money_get(size_t refs = 0);

// Les méthodes de lecture des montants :
    iter_type get(iter_type s, iter_type end, bool intl,
        ios_base &f, ios_base::iostate &err,
        long double &units) const;
    iter_type get(iter_type s, iter_type end, bool intl,
        ios_base &f, ios_base::iostate &err,
        string_type &digits) const;
    static const bool intl = Intl;

// L'identificateur de la facette :
```

```
    static locale::id id;
};
```

> **Note :** Les méthodes virtuelles d'implémentation des méthodes publiques n'ont pas été écrites dans la déclaration précédente par souci de simplification. Elles existent malgré tout, et peuvent être redéfinies par les classes dérivées afin de personnaliser le comportement de la facette.

Comme vous pouvez le constater, les méthodes d'écriture et de lecture `put` et `get` de ces facettes sont semblables aux méthodes correspondantes des facettes de gestion des nombres. Toutefois, elles se distinguent par un paramètre booléen complémentaire qui permet d'indiquer si les opérations de formatage doivent se faire en utilisant les conventions internationales d'écriture des montants. Les autres paramètres ont la même signification que pour les méthodes `put` et `get` des facettes de gestion des nombres. En particulier, l'itérateur fourni indique l'emplacement où les données doivent être écrites ou lues, et le flux d'entrée / sortie spécifié permet de récupérer les options de formatage des montants. L'une des options les plus utiles est sans doute l'option qui permet d'afficher la base des nombres, car, dans le cas des facettes de gestion des montants, elle permet d'activer ou non l'écriture du symbole monétaire. Enfin, les méthodes `put` et `get` sont fournies en deux exemplaires, un pour chaque type de donnée utilisable pour représenter les montants, à savoir les double et les chaînes de caractères.

16.2.6. Les facettes de gestion du temps

La bibliothèque standard ne fournit que deux facettes pour l'écriture et la lecture des dates : la facette time_put et la facette time_get. Ces deux facettes utilisent le type de base struct tm de la bibliothèque C pour représenter le temps. Bien que ce document ne décrive pas les fonctions de la bibliothèque C, il est peut-être bon de rappeler comment les programmes C manipulent les dates en général.

La gestion du temps dans un programme peut très vite devenir un véritable cauchemar, principalement en raison de la complexité que les êtres humains se sont efforcés de développer dans leur manière de représenter le temps. En effet, il faut tenir compte non seulement des spécificités des calendriers (années bissextiles ou non par exemple), mais aussi des multiples bases de numération utilisées dans l'écriture des dates (24 heures par jour, 60 minutes par heure et 60 secondes par minutes, puis 10 dixièmes dans une seconde et ainsi de suite) et des conventions locales de gestion des heures (fuseau horaire, heure d'été et d'hiver). La règle d'or lors de la manipulation des dates est donc de toujours travailler dans un référentiel unique avec une représentation linéaire du temps, autrement dit, de simplifier tout cela. En pratique, cela revient à dire que les programmes doivent utiliser une représentation linéaire du temps (généralement, le nombre de secondes écoulées depuis une date de référence) et travailler en temps universel. De même, le stockage des dates doit être fait dans ce format afin de garantir la possibilité d'échanger les données sans pour autant laisser la place aux erreurs d'interprétation de ces dates.

En pratique, la bibliothèque C utilise le type time_t. Les valeurs de ce type représentent le nombre d'instants écoulés depuis le premier janvier 1970 à 0 heure (date considérée comme le début de l'ère informatique par les inventeurs du langage C, Kernighan et Ritchie, et que l'on appelle couramment « Epoch »). La durée de ces instants n'est pas normalisée par la bibliothèque C, mais il s'agit de secondes pour les systèmes POSIX. Le type time_t permet donc de réaliser des calculs simplement sur les dates. Les dates représentées avec des time_t sont toujours exprimées en temps universel.

Bien entendu, il existe des fonctions permettant de convertir les dates codées sous la forme de time_t en dates humaines et réciproquement. Le type de donnée utilisé pour stocker les dates au format humain est la structure struct tm. Cette structure contient plusieurs champs, qui représentent entre autres l'année, le jour, le mois, les heures, les minutes et les secondes. Ce type contient donc les dates au format éclaté et permet d'obtenir les différentes composantes d'une date.

Généralement, les dates sont formatées en temps local, car les utilisateurs désirent souvent avoir les dates affichées dans leur propre base de temps. Cependant, il est également possible de formater les dates en temps universel. Ces opérations de formatages sont réalisées par les bibliothèques C et C++, et les programmes n'ont donc pas à se soucier des paramètres de fuseaux horaires, d'heure d'été et d'hiver et des conventions locales d'écriture des dates : tout est pris en charge par les locales.

Les principales fonctions permettant de manipuler les dates sont récapitulées dans le tableau ci-dessous :

Tableau 16-1. Fonctions C de gestion des dates

Fonction	Description
`time_t time(time_t *)`	Permet d'obtenir la date courante. Peut être appelée avec l'adresse d'une variable de type time_t en paramètre ou avec la constante `NULL`. Initialise la variable passée par pointeur avec la date courante, et renvoie également la valeur écrite.
`struct tm *gmtime(const time_t *)`	Permet de convertir une date stockée dans une variable de type time_t en sa version éclatée en temps universel. Le pointeur renvoyé référence une structure allouée en zone statique par la bibliothèque C et ne doit pas être libéré.
`struct tm *localtime(const time_t *)`	Permet de convertir une date stockée dans une variable de type time_t en sa version éclatée en temps local. Le pointeur renvoyé référence une structure allouée en zone statique par la bibliothèque C et ne doit pas être libéré.
`time_t mktime(struct tm *)`	Permet de construire une date de type time_t à partir d'une date en temps local stockée dans une structure struct tm. Les données membres de la structure struct tm peuvent être corrigées par la fonction `mktime` si besoin est. Cette fonction est donc la fonction inverse de `localtime`.
`size_t strftime(char *tampon, size_t max, const char *format, constr struct tm *t)`	Permet de formater une date stockée dans une structure struct tm dans une chaîne de caractères. Cette chaîne doit être fournie en premier paramètre, ainsi que le nombre maximal de caractères que la fonction pourra écrire. La fonction renvoie le nombre de caractères écrits ou, si le premier paramètre est nul, la taille de la chaîne de caractères qu'il faudrait pour effectuer une écriture complète. La fonction `strftime` prend en paramètre une chaîne de format et fonctionne de manière similaire aux fonctions `printf` et `sprintf`. Elle comprend un grand nombre de formats, mais les plus utiles sont sans doute les formats « `%X` » et « `%x` », qui permettent respectivement de formater l'heure et la date selon les conventions de la locale du programme.

Note : Il n'existe pas de fonction permettant de convertir une date exprimée en temps universel et stockée dans une structure struct tm en une date de type time_t. De même, la bibliothèque C ne fournit pas de fonction permettant d'analyser une chaîne de caractères représentant une date. Cependant, la norme Unix 98 définit la fonction `strptime`, qui est la fonction inverse de la

fonction `strftime`.

Les fonctions `localtime` et `gmtime` ne sont pas sûres dans un environnement multithreadé. En effet, la zone de mémoire renvoyée est en zone statique et est partagée par tous les threads. La bibliothèque C définit donc deux fonctions complémentaires, `localtime_r` et `gmtime_r`, qui prennent un paramètre complémentaire qui doit recevoir un pointeur sur la structure struct tm dans lequel le résultat doit être écrit. Cette structure est allouée par le thread appelant et ne risque donc pas d'être détruite par un appel à la même fonction par un autre thread.

Les facettes de la bibliothèque standard C++ ne permettent pas de manipuler les dates en soi. Elles ne sont destinées qu'à réaliser le formatage des dates en tenant compte des spécificités de représentation des dates de la locale. Elles se comportent exactement comme la fonction `strftime` le fait lorsque l'on utilise les chaînes de format « `%X` » et « `%x` ».

16.2.6.1. La facette d'écriture des dates

La facette d'écriture des dates est déclarée comme suit dans l'en-tête `locale` :

```
template <class charT,
    class OutputIterator = ostreambuf_iterator<charT> >
class time_put : public locale::facet
{
public:
// Les types de données :
    typedef charT          char_type;
    typedef OutputIterator iter_type;

// Le constructeur :
    explicit time_put(size_t refs = 0);

// Les méthodes d'écriture des dates :
    iter_type put(iter_type s, ios_base &f, char_type remplissage, const tm *t,
        char format, char modificateur = 0) const;
    iter_type put(iter_type s, ios_base &f, char_type remplissage, const tm *t,
        const charT *debut_format, const charT *fin_format) const;

// L'identificateur de la facette :
      static locale::id id;
};
```

Note : Les méthodes virtuelles d'implémentation des méthodes publiques n'ont pas été écrites dans la déclaration précédente par souci de simplification. Elles existent malgré tout, et peuvent être redéfinies par les classes dérivées afin de personnaliser le comportement de la facette.

Cette facette dispose de deux surcharges de la méthode `put` permettant d'écrire une date sur un flux de sortie. La première permet d'écrire une date sur le flux de sortie dont un itérateur est donné en premier paramètre. Le formatage de la date se fait comme avec la fonction `strftime` de la bibliothèque C. Le paramètre `modificateur` ne doit pas être utilisé en général, sa signification n'étant pas précisée par la norme C++. La deuxième forme de la méthode `put` réalise également une écriture sur le flux, en prenant comme chaîne de format la première sous-chaîne commençant par le caractère '`%`' dans la chaîne indiquée par les paramètres `debut_format` et `fin_format`.

16.2.6.2. La facette de lecture des dates

La facette de lecture des dates permet de lire les dates dans le même format que celui utilisé par la fonction `strftime` de la bibliothèque C lorsque la chaîne de format vaut « `%X` » ou « `%x` ». Cette facette est déclarée comme suit dans l'en-tête `locale` :

```
template <class charT,
    class InputIterator = istreambuf_iterator<charT> >
class time_get : public locale::facet, public time_base
{
public:
// Les types de données :
    typedef charT         char_type;
    typedef InputIterator iter_type;

// Le constructeur :
    explicit time_get(size_t refs = 0);

// Les méthodes de gestion de la lecture des dates :
    iter_type get_time(iter_type s, iter_type end, ios_base &f,
        ios_base::iostate &err, tm *t)  const;
    iter_type get_date(iter_type s, iter_type end, ios_base &f,
        ios_base::iostate &err, tm *t)  const;
    iter_type get_weekday(iter_type s, iter_type end, ios_base &f,
        ios_base::iostate &err, tm *t) const;
    iter_type get_monthname(iter_type s, iter_type end, ios_base &f,
        ios_base::iostate &err, tm *t) const;
    iter_type get_year(iter_type s, iter_type end, ios_base &f,
        ios_base::iostate &err, tm *t) const;
    dateorder date_order() const;

// L'identificateur de la facette :
    static locale::id id;
};
```

> **Note :** Les méthodes virtuelles d'implémentation des méthodes publiques n'ont pas été écrites dans la déclaration précédente par souci de simplification. Elles existent malgré tout, et peuvent être redéfinies par les classes dérivées afin de personnaliser le comportement de la facette.

Les différentes méthodes de cette facette permettent respectivement d'obtenir l'heure, la date, le jour de la semaine, le nom du mois et l'année d'une date dans le flux d'entrée spécifié par l'itérateur fourni en premier paramètre. Toutes ces données sont interprétées en fonction de la locale à laquelle la facette appartient.

Enfin, la méthode `date_order` permet d'obtenir l'une des valeurs de l'énumération définie dans la classe de base time_base et qui indique l'ordre dans lequel les composants jour / mois / année des dates apparaissent dans la locale de la facette. La classe de base time_base est déclarée comme suit dans l'en-tête `locale` :

```
class time_base
{
public:
    enum dateorder
    {
```

```
        no_order, dmy, mdy, ymd, ydm
    };
};
```

La signification des différentes valeurs de l'énumération est immédiate. La seule valeur nécessitant des explications complémentaires est la valeur `no_order`. Cette valeur est renvoyée par la méthode `date_order` si le format de date utilisé par la locale de la facette contient d'autres champs que le jour, le mois et l'année.

> **Note :** La méthode `date_order` est fournie uniquement à titre de facilité par la bibliothèque standard. Elle peut ne pas être implémentée pour certaines locales. Dans ce cas, elle renvoie systématiquement la valeur `no_order`.

16.2.7. Les facettes de gestion des messages

Afin de faciliter l'internationalisation des programmes, la bibliothèque standard fournit la facette messages, qui permet de prendre en charge la traduction de tous les messages d'un programme de manière indépendante du système sous-jacent. Cette facette permet d'externaliser tous les messages des programmes dans des fichiers de messages que l'on appelle des *catalogues*. Le format et l'emplacement de ces fichiers ne sont pas spécifiés par la norme C++, cependant, la manière d'y accéder est standardisée et permet d'écrire des programmes portables. Ainsi, lorsqu'un programme devra être traduit, il suffira de traduire les messages stockés dans les fichiers de catalogue pour chaque langue et de les distribuer avec le programme.

> **Note :** La manière de créer et d'installer ces fichiers étant spécifique à chaque implémentation de la bibliothèque standard et, dans une large mesure, spécifique au système d'exploitation utilisé, ces fichiers ne seront pas décrits ici. Seule la manière d'utiliser la facette messages sera donc indiquée. Reportez-vous à la documentation de votre environnement de développement pour plus de détails sur les outils permettant de générer les fichiers de catalogue.

La facette messages référence les fichiers de catalogue à l'aide d'un type de donnée spécifique. Ce type de donnée est défini dans la classe de base messages_base comme étant un type intégral :

```
class messages_base
{
public:
    typedef int catalog;
};
```

La classe `template` messages de gestion de la facette hérite donc de cette classe de base et utilise le type catalog pour identifier les fichiers de catalogue de l'application.

La classe messages est déclarée comme suit dans l'en-tête `locale` :

```
template <class charT>
class messages : public locale::facet, public messages_base
{
public:
// Les types de données :
```

```
    typedef charT char_type;
    typedef basic_string<charT> string_type;

// Le constructeur :
    explicit messages(size_t refs = 0);

// Les méthodes de gestion des catalogues de messages :
    catalog open(const basic_string<char> &nom, const locale &l) const;
    void    close(catalog c) const;
    string_type  get(catalog c, int groupe, int msg,
        const string_type &defaut) const;

// L'identificateur de la facette :
    static locale::id id;
};
```

> **Note :** Les méthodes virtuelles d'implémentation des méthodes publiques n'ont pas été écrites dans la déclaration précédente par souci de simplification. Elles existent malgré tout, et peuvent être redéfinies par les classes dérivées afin de personnaliser le comportement de la facette.

Les principales méthodes de gestion des catalogues sont les méthodes `open` et `close`. Comme leurs noms l'indiquent, ces méthodes permettent d'ouvrir un nouveau fichier de catalogue et de le fermer pour en libérer les ressources. La méthode `open` prend en paramètre le nom du catalogue à ouvrir. Ce nom doit identifier de manière unique le catalogue, mais la norme C++ n'indique pas comment il doit être interprété. Cela relève donc de l'implémentation de la bibliothèque standard utilisée. Toutefois, en pratique, il est probable qu'il s'agit d'un nom de fichier. Le deuxième paramètre permet d'indiquer la locale à utiliser pour effectuer les conversions de jeux de caractères si cela est nécessaire. Il permet donc de laisser au programmeur le choix du jeu de caractères dans lequel les messages seront écrits dans le catalogue. La valeur renvoyée par la méthode `open` est l'identifiant du catalogue, identifiant qui devra être fourni à la méthode `get` pour récupérer les messages du catalogue et à la méthode `close` pour fermer le fichier de catalogue. Si l'ouverture du fichier n'a pas pu être effectuée, la méthode `open` renvoie une valeur inférieure à 0.

Les messages du catalogue peuvent être récupérés à l'aide de la méthode `get`. Cette méthode prend en paramètre l'identifiant d'un catalogue précédemment obtenu par l'intermédiaire de la méthode `open`, un identifiant de groupe de message et un identifiant d'un message. Le dernier paramètre doit recevoir la valeur par défaut du message en cas d'échec de la recherche du message dans le catalogue. Cette valeur par défaut est souvent un message en anglais, ce qui permet au programme de fonctionner correctement même lorsque ses fichiers de catalogue sont vides.

La manière dont les messages sont identifiés n'est pas spécifiée par la norme C++, tout comme la manière dont ils sont classés en groupes de messages au sein d'un même fichier de catalogue. Cela relève donc de l'implémentation de la bibliothèque utilisée. Consultez la documentation de votre environnement de développement pour plus de détails à ce sujet.

> **Note :** Cette facette est relativement peu utilisée, pour plusieurs raison. Premièrement, peu d'environnements C++ respectent la norme C++ à ce jour. Deuxièmement, les systèmes d'exploitation disposent souvent de mécanismes de localisation performants et pratiques. Enfin, l'identification d'un message par des valeurs numériques n'est pas toujours pratique et il est courant d'utiliser le message par défaut, souvent en anglais, comme clef de recherche pour les messages internationaux. Cette manière de procéder est en effet beaucoup plus simple,

puisque le contenu des messages est écrit en clair dans la langue par défaut dans les fichiers sources du programme.

16.3. Personnalisation des mécanismes de localisation

Les mécanismes de localisation ont été conçus de telle sorte que le programmeur peut, s'il le désire (et s'il en a réellement le besoin), personnaliser leur fonctionnement. Ainsi, il est parfaitement possible de définir de nouvelles facettes, par exemple pour permettre la localisation des types de données complémentaires définis par le programme. De même, il est possible de redéfinir les méthodes virtuelles des classes de gestion des facettes standards de la bibliothèque et de remplacer les facettes originales par des facettes personnalisées. Cependant, il faut bien reconnaître que la manière de procéder n'est pas très pratique, et en fait les mécanismes internes de gestion des facettes semblent être réservés aux classes et aux méthodes de la bibliothèque standard elle-même.

16.3.1. Création et intégration d'une nouvelle facette

Comme il l'a été expliqué dans la Section 16.1, une facette n'est rien d'autre qu'une classe dérivant de la classe locale::facet et contenant une donnée membre statique `id`. Cette donnée membre est utilisée par les classes de locale pour identifier le type de la facette et pour l'intégrer dans le mécanisme de gestion des facettes standards.

L'exemple suivant montre comment on peut réaliser deux facettes permettant d'encapsuler les spécificités d'un type de donnée défini par le programme, le type answer_t. Ce type est supposé permettre la création de variables contenant la réponse de l'utilisateur à une question. Ce n'est rien d'autre qu'une énumération contenant les valeurs `no` (pour la réponse négative), `yes` (pour l'affirmative), `all` (pour répondre par l'affirmative pour tout un ensemble d'éléments) et `none` (pour répondre par la négative pour tout un ensemble d'éléments).

Dans cet exemple, deux facettes sont définies : la facette answerpunct, qui prend en charge la localisation des noms des différentes valeurs de l'énumération answer_t, et la facette answer_put, qui prend en charge le formatage des valeurs de cette énumération dans un flux standard. L'opérateur `operator<<` est également défini, afin de présenter la manière dont ces facettes peuvent être utilisées. La facette answer_get et l'opérateur correspondant `operator>>` n'ont pas été définis et sont laissés en exercice pour le lecteur intéressé.

Exemple 16-6. Définition de nouvelles facettes

```
#include <iostream>
#include <locale>

using namespace std;

// Nouveau type de donnée permettant de gérer les réponses
// aux questions (yes / no / all / none) :
enum answer_t
{
    no, yes, all, none
};

// Facette prenant définissant les noms des réponses :
```

```
template <class charT>
class answerpunct : public locale::facet
{
public:
    // Les types de données :
    typedef charT char_type;
    typedef basic_string<charT> string_type;

    // L'identifiant de la facette :
    static locale::id id;

    // Le constructeur :
    answerpunct(size_t refs = 0) : locale::facet(refs)
    {
    }

    // Les méthodes permettant d'obtenir les noms des valeurs :
    string_type yesname() const
    {
        return do_yesname();
    }

    string_type noname() const
    {
        return do_noname();
    }

    string_type allname() const
    {
        return do_allname();
    }

    string_type nonename() const
    {
        return do_nonename();
    }

protected:
    // Le destructeur :
    virtual ~answerpunct()
    {
    }

    // Les méthodes virtuelles :
    virtual string_type do_yesname() const
    {
        return "yes";
    }

    virtual string_type do_noname() const
    {
        return "no";
    }

    virtual string_type do_allname() const
    {
        return "all";
```

```
    }

    virtual string_type do_nonename() const
    {
        return "none";
    }
};

// Instanciation de l'identifiant de la facette answerpunct :
template <class charT>
locale::id answerpunct<charT>::id;

// Facette prenant en charge le formatage des réponses :
template <class charT,
    class OutputIterator = ostreambuf_iterator<charT> >
class answer_put : public locale::facet
public:
    // Les types de données :
    typedef charT char_type;
    typedef OutputIterator      iter_type;
    typedef basic_string<charT> string_type;

    // L'identifiant de la facette :
    static locale::id id;

    // Le constructeur :
    answer_put(size_t refs = 0) : locale::facet(refs)
    {
    }

    // La méthode de formatage publique :
    iter_type put(iter_type i, ios_base &flux,
        char_type remplissage, answer_t valeur) const
    {
        return do_put(i, flux, remplissage, valeur);
    }

protected:
    // Le destructeur :
    virtual ~answer_put()
    {
    }

    // L'implémentation de la méthode de formatage :
    virtual iter_type do_put(iter_type i, ios_base &flux,
        char_type remplissage, answer_t valeur) const
    {
        // Récupère la facette décrivant les noms de types :
        const answerpunct<charT> &facet =
            use_facet<answerpunct<charT> >(flux.getloc());
        // Récupération du nom qui sera écrit :
        string_type result;
        switch (valeur)
        {
        case yes:
            result = facet.yesname();
            break;
```

```
            case no:
                result = facet.noname();
                break;
            case all:
                result = facet.allname();
                break;
            case none:
                result = facet.nonename();
                break;
            }
            // Écriture de la valeur :
            const char *p = result.c_str();
            while (*p != 0)
            {
                *i = *p;
                ++i; ++p;
            }
            return i;
        }
};

// Instanciation de l'identifiant de la facette answer_put :
template <class charT,
    class OutputIterator = ostreambuf_iterator<charT> >
locale::id answer_put<charT, OutputIterator>::id;

// Opérateur permettant de formater une valeur
// de type answer_t dans un flux de sortie :
template <class charT, class Traits>
basic_ostream<charT, Traits> &operator<<(
    basic_ostream<charT, Traits> &flux,
    answer_t valeur)
{
    // Initialisation du flux de sortie :
    typename basic_ostream<charT, Traits>::sentry init(flux);
    if (init)
    {
        // Récupération de la facette de gestion de ce type :
        const answer_put<charT> &facet =
            use_facet<answer_put<charT> >(flux.getloc());
        // Écriture des données :
        facet.put(flux, flux, ' ', valeur);
    }
    return flux;
}

int main(void)
{
    // Crée une nouvelle locale utilisant nos deux facettes :
    locale temp(locale(""), new answerpunct<char>);
    locale loc(temp, new answer_put<char>);
    // Installe cette locale dans le flux de sortie :
    cout.imbue(loc);
    // Affiche quelques valeurs de type answer_t :
    cout << yes << endl;
    cout << no << endl;
    cout << all << endl;
```

```
    cout << none << endl;
    return 0;
}
```

Note : Cet exemple, bien que déjà compliqué, passe sous silence un certain nombre de points qu'il faudrait théoriquement prendre en compte pour réaliser une implémentation correcte des facettes et des opérateurs d'insertion et d'extraction des données de type answer_t dans les flux standards. Il faudrait en effet traiter les cas d'erreurs lors des écritures sur le flux de sortie dans la méthode `do_put` de la facette answer_put, capter les exceptions qui peuvent se produire, corriger l'état du flux d'entrée / sortie au sein de l'opérateur `operator<<` et relancer ces exceptions.

De même, les paramètres de la locale ne sont absolument pas pris en compte dans la facette answerpunct, alors qu'une implémentation complète devrait s'en soucier. Pour cela, il faudrait récupérer le nom de la locale incluse dans les flux d'entrée / sortie d'une part, et définir une facette spécialisée answerpunct_byname, en fonction du nom de laquelle les méthodes `do_yesname`, `do_noname`, `do_allname` et `do_nonename` devraient s'adapter. La section suivante donne un exemple de redéfinition d'une facette existante.

16.3.2. Remplacement d'une facette existante

La redéfinition des méthodes de facettes déjà existantes est légèrement plus simple que l'écriture d'une nouvelle facette. En effet, il n'est plus nécessaire de définir la donnée membre statique *id*. De plus, seules les méthodes qui doivent réellement être redéfinies doivent être récrites.

L'exemple suivant présente comment un programme peut redéfinir les méthodes `do_truename` et `do_falsename` de la facette standard `numpunct_byname` afin d'en fournir une version localisée en français. Cela permet d'utiliser ces noms français dans les opérations de formatage des flux d'entrée / sortie standards, lorsque le manipulateur `boolalpha` a été utilisé.

Exemple 16-7. Spécialisation d'une facette existante

```
#include <iostream>
#include <locale>
#include <clocale>
#include <cstring>

using namespace std;

// Facette destinée à remplacer numpunct_byname :
class MyNumpunct_byname :
    public numpunct_byname<char>
{
    // Les noms des valeurs true et false :
    const char *m_truename;
    const char *m_falsename;

public:
    MyNumpunct_byname(const char* nom) :
        numpunct_byname<char>(nom)
    {
        // Détermine le nom de la locale active :
        const char *loc = nom;
        if (strcmp(nom, "") == 0)
```

```
            {
                // Récupère le nom de la locale globale active :
                loc = setlocale(0, NULL);
            }
            // Prend en charge les noms français :
            if (strcmp(loc, "fr_FR") == 0)
            {
                m_truename = "vrai";
                m_falsename = "faux";
            }
            else
            {
                // Pour les autres locales, utilise les noms anglais :
                m_truename = "true";
                m_falsename = "false";
            }
        }

    protected:
        ~MyNumpunct_byname()
        {
        }

        string do_truename() const
        {
            return m_truename;
        }

        string do_falsename() const
        {
            return m_falsename;
        }
    };

    int main(void)
    {
        // Fixe la locale globale du programme :
        locale::global(locale(""));
        // Crée une nouvelle locale utilisant notre facette :
           locale l(locale(""), new MyNumpunct_byname(""));
        // Installe cette locale dans le flux de sortie :
        cout.imbue(l);
        // Affiche deux booléens :
        cout << boolalpha << true << endl;
        cout << false << endl;
        return 0;
    }
```

Note : La classe de base de la facette MyNumpunct_byname est la classe `numpunct_byname` parce que la facette a besoin de connaître le nom de la locale pour laquelle elle est construite. En effet, aucun autre mécanisme standard ne permet à une facette de récupérer ce nom et donc de s'adapter aux différentes locales existantes. Vous remarquerez que les facettes de formatage n'ont pas besoin de connaître ce nom puisqu'elles peuvent le récupérer grâce à la méthode `name` de la locale du flux sur lequel elles travaillent.

La facette MyNumpunct_byname utilise la fonction `setlocale` de la bibliothèque C pour récupérer le nom de la locale courante si elle est initialisée avec un nom vide. En réalité, elle devrait

récupérer ce nom par ses propres moyens et effectuer les traductions des noms des valeurs `true` et `false` par elle-même, car cela suppose que la locale globale du programme est initialisée avec le même nom. C'est pour cela que le programme principal commence par appeler la méthode `global` de la classe local avec comme paramètre une locale anonmyme. Cela dit, les mécanismes permettant à un programme de récupérer les paramètres de la locale définie dans l'environnement d'exécution du programme sont spécifiques à chaque système et ne peuvent donc pas être décrits ici.

Bien entendu, si d'autres langues que le français devaient être prises en compte, d'autre mécanismes plus génériques devraient également être mis en place pour définir les noms des valeurs `true` et `false` afin d'éviter de compliquer exagérément le code de la facette.

Chapitre 17. Les conteneurs

La plupart des programmes informatiques doivent, à un moment donné ou à un autre, conserver un nombre arbitraire de données en mémoire, généralement pour y accéder ultérieurement et leur appliquer des traitements spécifiques. En général, les structures de données utilisées sont toujours manipulées par des algorithmes classiques, que l'on retrouve donc souvent, si ce n'est plusieurs fois, dans chaque programme. Ces structures de données sont communément appelées des *conteneurs* en raison de leur capacité à contenir d'autres objets.

Afin d'éviter aux programmeurs de réinventer systématiquement la roue et de reprogrammer les structures de données et leurs algorithmes associés les plus classiques, la bibliothèque standard définit un certain nombre de classes `template` pour les conteneurs les plus courants. Ces classes sont paramétrées par le type des données des conteneurs et peuvent donc être utilisées virtuellement pour toutes les situations qui se présentent.

Les conteneurs de la bibliothèque standard ne sont pas définis par les algorithmes qu'ils utilisent, mais plutôt par l'interface qui peut être utilisée par les programmes clients. La bibliothèque standard impose également des contraintes de performances sur ces interfaces en termes de complexité. En réalité, ces contraintes sont tout simplement les plus fortes qui soient, ce qui garantit aux programmes qui les utilisent qu'ils auront les meilleures performances possibles.

La bibliothèque classifie les conteneurs en deux grandes catégories selon leurs fonctionnalités : les *séquences* et les *conteneurs associatifs*. Une séquence est un conteneur capable de stocker ses éléments de manière séquentielle, les uns à la suite des autres. Les éléments sont donc parfaitement identifiés par leur position dans la séquence, et leur ordre relatif est donc important. Les conteneurs associatifs, en revanche, manipulent leurs données au moyen de valeurs qui les identifient indirectement. Ces identifiants sont appelées des *clefs* par analogie avec la terminologie utilisée dans les bases de données. L'ordre relatif des éléments dans le conteneur est laissé dans ce cas à la libre discrétion de ce dernier et leur recherche se fait donc, généralement, par l'intermédiaire de leurs clefs.

La bibliothèque fournit plusieurs conteneurs de chaque type. Chacun a ses avantages et ses inconvénients. Comme il n'existe pas de structure de données parfaite qui permette d'obtenir les meilleures performances sur l'ensemble des opérations réalisables, l'utilisateur des conteneurs de la bibliothèque standard devra effectuer son choix en fonction de l'utilisation qu'il désire en faire. Par exemple, certains conteneurs sont plus adaptés à la recherche d'éléments mais sont relativement coûteux pour les opérations d'insertion ou de suppression, alors que pour d'autres conteneurs, c'est exactement l'inverse. Le choix des conteneurs à utiliser sera donc déterminant quant aux performances finales des programmes.

17.1. Fonctionnalités générales des conteneurs

Au niveau de leurs interfaces, tous les conteneurs de la bibliothèque standard présentent des similitudes. Cet état de fait n'est pas dû au hasard, mais bel et bien à la volonté de simplifier la vie des programmeurs en évitant de définir une multitude de méthodes ayant la même signification pour chaque conteneur. Cependant, malgré cette volonté d'uniformisation, il existe des différences entre les différents types de conteneurs (séquences ou conteneurs associatifs). Ces différences proviennent essentiellement de la présence d'une clef dans ces derniers, qui permet de manipuler les objets contenus plus facilement.

Quelle que soit leur nature, les conteneurs fournissent un certain nombre de services de base que le programmeur peut utiliser. Ces services comprennent la définition des itérateurs, de quelques types complémentaires, des opérateurs et de fonctions standards. Les sections suivantes vous présentent ces

fonctionnalités générales. Toutefois, les descriptions données ici ne seront pas détaillées outre mesure car elles seront reprises en détail dans la description de chaque conteneur.

17.1.1. Définition des itérateurs

Pour commencer, il va de soi que tous les conteneurs de la bibliothèque standard disposent d'itérateurs. Comme on l'a vu dans la Section 13.4, les itérateurs constituent une abstraction de la notion de pointeur pour les tableaux. Ils permettent donc de parcourir tous les éléments d'un conteneur séquentiellement à l'aide de l'opérateur de déréférencement `*` et de l'opérateur d'incrémentation `++`.

Les conteneurs définissent donc tous un type iterator et un type const_iterator, qui sont les types des itérateurs sur les éléments du conteneur. Le type d'itérateur const_iterator est défini pour accéder aux éléments d'un conteneur en les considérant comme des constantes. Ainsi, si le type des éléments stockés dans le conteneur est T, le déréférencement d'un const_iterator renverra un objet de type const T.

Les conteneurs définissent également les types de données difference_type et size_type que l'on peut utiliser pour effectuer des calculs d'arithmétique des pointeurs avec leurs itérateurs. Le type difference_type se distingue du type size_type par le fait qu'il peut contenir toute valeur issue de la différence entre deux itérateurs, et accepte donc les valeurs négatives. Le type size_type quant à lui est utilisé plus spécialement pour compter un nombre d'éléments, et ne peut prendre que des valeurs positives.

Afin de permettre l'initialisation de leurs itérateurs, les conteneurs fournissent deux méthodes `begin` et `end`, qui renvoient respectivement un itérateur référençant le premier élément du conteneur et la valeur de fin de l'itérateur, lorsqu'il a passé le dernier élément du conteneur. Ainsi, le parcours d'un conteneur se fait typiquement de la manière suivante :

```
// Obtient un itérateur sur le premier élément :
Conteneur::iterateur i = instance.begin();
// Boucle sur toutes les valeurs de l'itérateur
// jusqu'à la dernière :
while (i != instance.end())
{
    // Travaille sur l'élément référencé par i :
    f(*i);
    // Passe à l'élément suivant :
    ++i;
}
```

où `Conteneur` est la classe de du conteneur et `instance` en est une instance.

> **Note :** Pour des raisons de performances et de portabilité, la bibliothèque standard ne fournit absolument aucun support du multithreading sur ses structures de données. En fait, la gestion du multithreading est laissée à la discrétion de chaque implémentation. Généralement, seul le code généré par le compilateur est sûr vis-à-vis des threads (en particulier, les opérateurs d'allocation mémoire `new` et `new[]`, ainsi que les opérateurs `delete` et `delete[]` peuvent être appelés simultanément par plusieurs threads pour des objets différents). Il n'en est pas de même pour les implémentations des conteneurs et des algorithmes de la bibliothèque standard.
>
> Par conséquent, si vous voulez accéder à un conteneur à partir de plusieurs threads, vous devez prendre en charge vous-même la gestion des sections critiques afin de vous assurer que ce conteneur sera toujours dans un état cohérent. En fait, il est recommandé de le faire même si l'implémentation de la bibliothèque standard se protège elle-même contre les accès concurrents

à partir de plusieurs threads, afin de rendre vos programmes portables vers d'autres environnements.

Les itérateurs utilisés par les conteneurs sont tous au moins du type ForwardIterator. En pratique, cela signifie que l'on peut parcourir les itérateurs du premier au dernier élément, séquentiellement. Cependant, la plupart des conteneurs disposent d'itérateurs au moins bidirectionnels, et peuvent donc être parcourus dans les deux sens. Les conteneurs qui disposent de ces propriétés sont appelés des *conteneurs réversibles*.

Les conteneurs réversibles disposent, en plus des itérateurs directs, d'itérateurs inverses. Ces itérateurs sont repectivement de type reverse_iterator et const_reverse_iterator. Leur initialisation peut être réalisée à l'aide de la fonction `rbegin`, et leur valeur de fin peut être récupérée à l'aide de la fonction `rend`.

17.1.2. Définition des types de données relatifs aux objets contenus

Outre les types d'itérateurs, les conteneurs définissent également des types spécifiques aux données qu'ils contiennent. Ces types de données permettent de manipuler les données des conteneurs de manière générique, sans avoir de connaissance précises sur la nature réelle des objets qu'ils stockent. Ils sont donc couramment utilisés par les algorithmes de la bibliothèque standard.

Le type réellement utilisé pour stocker les objets dans un conteneur n'est pas toujours le type `template` utilisé pour instancier ce conteneur. En effet, certains conteneurs associatifs stockent les clefs des objets avec la valeur des objets eux-mêmes. Ils utilisent pour cela la classe pair, qui permet de stocker, comme on l'a vu en Section 14.2.2, des couples de valeurs. Le type des données stockées par ces conteneurs est donc plus complexe que le simple type `template` par lequel ils sont paramétrés.

Afin de permettre l'uniformisation des algorithmes travaillant sur ces types de données, les conteneurs définissent tous le type value_type dans leur classe `template`. C'est en particulier ce type qu'il faut utiliser lors des insertions d'éléments dans les conteneurs. Bien entendu, pour la plupart des conteneurs, et pour toutes les séquences, le type value_type est effectivement le même type que le type `template` par lequel les conteneurs sont paramétrés.

Les conteneurs définissent également d'autres types permettant de manipuler les données qu'ils stockent. En particulier, le type reference est le type des références sur les données, et le type const_reference est le type des références constantes sur les données. Ces types sont utilisés par les méthodes des conteneurs qui permettent d'accéder à leurs données.

17.1.3. Spécification de l'allocateur mémoire à utiliser

Toutes les classes `template` des conteneurs de la bibliothèque standard utilisent la notion d'allocateur pour réaliser les opérations de manipulation de la mémoire qu'elles doivent effectuer lors du stockage de leurs éléments ou lors de l'application d'algorithmes spécifiques au conteneur. Le type des allocateurs peut être spécifié dans la liste des paramètres `template` des conteneurs, en marge du type des données contenues. Les constructeurs des conteneurs prennent tous un paramètre de ce type, qui sera l'allocateur mémoire utilisé pour ce conteneur. Ainsi, il est possible de spécifier un allocateur spécifique pour chaque conteneur, qui peut être particulièrement optimisé pour le type des données gérées par ce conteneur.

Toutefois, le paramètre `template` spécifiant la classe de l'allocateur mémoire à utiliser dispose d'une valeur par défaut, qui représente l'allocateur standard de la bibliothèque allocator<T>. Il n'est donc pas nécessaire de spécifier cet allocateur lors de l'instanciation d'un conteneur. Cela rend plus simple l'utilisation de la bibliothèque standard C++ pour ceux qui ne désirent pas développer eux-même un allocateur mémoire. Par exemple, la déclaration `template` du conteneur list est la suivante :

```
template <class T, class Allocator = allocator<T> >
```

Il est donc possible d'instancier une liste d'entiers simplement en ne spécifiant que le type des objets contenus, en l'occurrence, des entiers :

```
typedef list<int> liste_entier;
```

De même, le paramètre des constructeurs permettant de spécifier l'allocateur à utiliser pour les conteneurs dispose systématiquement d'une valeur par défaut, qui est l'instance vide du type d'allocateur spécifié dans la liste des paramètres `template`. Par exemple, la déclaration du constructeur le plus simple de la classe list est la suivante :

```
template <class T, class Allocator>
list<T, Allocator>::list(const Allocator & = Allocator());
```

Il est donc parfaitement légal de déclarer une liste d'entier simplement de la manière suivante :

```
liste_entier li;
```

> **Note :** Il est peut-être bon de rappeler que toutes les instances d'un allocateur accèdent à la même mémoire. Ainsi, il n'est pas nécessaire, en général, de préciser l'instance de l'allocateur dans le constructeur des conteneurs. En effet, le paramètre par défaut fourni par la bibliothèque standard n'est qu'une instance parmi d'autres qui permet d'accéder à la mémoire gérée par la classe de l'allocateur fournie dans la liste des paramètres `template`.

Si vous désirez spécifier une classe d'allocateur différente de celle de l'allocateur standard, vous devrez faire en sorte que cette classe implémente toutes les méthodes des allocateurs de la bibliothèque standard. La notion d'allocateur a été détaillée dans la Section 13.6.

17.1.4. Opérateurs de comparaison des conteneurs

Les conteneurs disposent d'opérateurs de comparaison permettant d'établir des relations d'équivalence ou des relations d'ordre entre eux.

Les conteneurs peuvent tous être comparés directement avec les opérateurs == et !=. La relation d'égalité entre deux conteneurs est définie par le respect des deux propriétés suivantes :

- les deux conteneurs doivent avoir la même taille ;
- leurs éléments doivent être identiques deux à deux.

Si le type des objets contenus dispose des opérateurs d'infériorité et de supériorités strictes, les mêmes opérateurs seront également définis pour le conteneur. Ces opérateurs utilisent l'ordre lexicographique

pour déterminer le classement entre deux conteneurs. Autrement dit, l'opérateur d'infériorité compare les éléments des deux conteneurs un à un, et fixe son verdict dès la première différence constatée. Si un conteneur est un sous-ensemble du deuxième, le conteneur le plus petit est celui qui est inclus dans l'autre.

> **Note :** Remarquez que la définition des opérateurs de comparaison d'infériorité et de supériorité existe quel que soit le type des données que le conteneur peut stocker. Cependant, comme les conteneurs sont définis sous la forme de classes `template`, ces méthodes ne sont instanciées que si elles sont effectivement utilisées dans les programmes. Ainsi, il est possible d'utiliser les conteneurs même sur des types de données pour lesquels les opérateurs d'infériorité et de supériorité ne sont pas définis. Cependant, cette utilisation provoquera une erreur de compilation, car le compilateur cherchera à instancier les opérateurs à ce moment.

17.1.5. Méthodes d'intérêt général

Enfin, les conteneurs disposent de méthodes générales permettant d'obtenir des informations sur leurs propriétés. En particulier, le nombre d'éléments qu'ils contiennent peut être déterminé grâce à la méthode `size`. La méthode `empty` permet de déterminer si un conteneur est vide ou non. La taille maximale que peut prendre un conteneur est indiquée quant à elle par la méthode `max_size`. Pour finir, tous les conteneurs disposent d'une méthode `swap`, qui prend en paramètre un autre conteneur du même type et qui réalise l'échange des données des deux conteneurs. On utilisera de préférence cette méthode à toute autre technique d'échange car seules les références sur les structures de données des conteneurs sont échangées avec cette fonction, ce qui garantit une complexité indépendante de la taille des conteneurs.

17.2. Les séquences

Les séquences sont des conteneurs qui ont principalement pour but de stocker des objets afin de les traiter dans un ordre bien défini. Du fait de l'absence de clef permettant d'identifier les objets qu'elles contiennent, elles ne disposent d'aucune fonction de recherche des objets. Les séquences disposent donc généralement que des méthodes permettant de réaliser l'insertion et la suppression d'éléments, ainsi que le parcours des éléments dans l'ordre qu'elles utilisent pour les classer.

17.2.1. Fonctionnalités communes

Il existe un grand nombre de classes `template` de séquences dans la bibliothèque standard qui permettent de couvrir la majorité des besoins des programmeurs. Ces classes sont relativement variées tant dans leurs implémentations que dans leurs interfaces. Cependant, un certain nombre de fonctionnalités communes sont gérées par la plupart des séquences. Ce sont ces fonctionnalités que cette section se propose de vous décrire. Les fonctionnalités spécifiques à chaque classe de séquence seront détaillées séparément dans la Section 17.2.2.1.

Les exemples fournis dans cette section se baseront sur le conteneur list, qui est le type de séquence le plus simple de la bibliothèque standard. Cependant, ils sont parfaitement utilisables avec les autres types de séquences de la bibliothèque standard, avec des niveaux de performances éventuellement différents en fonction des séquences choisies bien entendu.

17.2.1.1. Construction et initialisation

La construction et l'initialisation d'une séquence peuvent se faire de multiples manières. Les séquences disposent en effet de plusieurs constructeurs et de deux surcharges de la méthode `assign` qui permet de leur affecter un certain nombre d'éléments. Le constructeur le plus simple ne prend aucun paramètre, hormis un allocateur standard à utiliser pour la gestion de la séquence, et permet de construire une séquence vide. Le deuxième constructeur prend en paramètre le nombre d'éléments initial de la séquence et la valeur de ces éléments. Ce constructeur permet donc de créer une séquence contenant déjà un certain nombre de copies d'un objet donné. Enfin, le troisième constructeur prend deux itérateurs sur une autre séquence d'objets qui devront être copiés dans la séquence en cours de construction. Ce constructeur peut être utilisé pour initialiser une séquence à partir d'une autre séquence ou d'un sous-ensemble de séquence.

Les surcharges de la méthode `assign` se comportent un peu comme les deux derniers constructeurs, à ceci près qu'elles ne prennent pas d'allocateur en paramètre. La première méthode permet donc de réinitialiser la liste et de la remplir avec un certain nombre de copies d'un objet donné, et la deuxième permet de réinitialiser la liste et de la remplir avec une séquence d'objets définie par deux itérateurs.

Exemple 17-1. Construction et initialisation d'une liste

```
#include <iostream>
#include <list>

using namespace std;

typedef list<int> li;

void print(li &l)
{
    li::iterator i = l.begin();
    while (i != l.end())
    {
        cout << *i << " " ;
        ++i;
    }
    cout << endl;
}

int main(void)
{
    // Initialise une liste avec trois éléments valant 5 :
    li l1(3, 5);
    print(l1);
    // Initialise une autre liste à partir de la première
    // (en fait on devrait appeler le constructeur de copie) :
    li l2(l1.begin(), l1.end());
    print(l2);
    // Affecte 4 éléments valant 2 à l1 :
    l1.assign(4, 2);
    print(l1);
    // Affecte l1 à l2 (de même, on devrait normalement
    // utiliser l'opérateur d'affectation) :
    l2.assign(l1.begin(), l1.end());
    print(l2);
    return 0;
}
```

Bien entendu, il existe également un constructeur et un opérateur de copie capables d'initialiser une séquence à partir d'une autre séquence du même type. Ainsi, il n'est pas nécessaire d'utiliser les constructeurs vus précédemment ni les méthodes `assign` pour initialiser une séquence à partir d'une autre séquence de même type.

17.2.1.2. Ajout et suppression d'éléments

L'insertion de nouveaux éléments dans une séquence se fait normalement à l'aide de l'une des surcharges de la méthode `insert`. Bien entendu, il existe d'autres méthodes spécifiques à chaque conteneur de type séquence et qui leur sont plus appropriées, mais ces méthodes ne seront décrites que dans les sections consacrées à ces conteneurs. Les différentes versions de la méthode `insert` sont récapitulées ci-dessous :

`iterator insert(iterator i, value_type valeur)`

Permet d'insérer une copie de la valeur spécifiée en deuxième paramètre dans le conteneur. Le premier paramètre est un itérateur indiquant l'endroit où le nouvel élément doit être inséré. L'insertion se fait immédiatement avant l'élément référencé par cet itérateur. Cette méthode renvoie un itérateur sur le dernier élément inséré dans la séquence.

`void insert(iterator i, size_type n, value_type valeur)`

Permet d'insérer `n` copies de l'élément spécifié en troisième paramètre avant l'élément référencé par l'itérateur `i` donné en premier paramètre.

`void insert(iterator i, iterator premier, iterator dernier)`

Permet d'insérer tous les éléments de l'intervalle défini par les itérateurs `premier` et `dernier` avant l'élément référencé par l'itérateur `i`.

Exemple 17-2. Insertion d'éléments dans une liste

```
#include <iostream>
#include <list>

using namespace std;

typedef list<int> li;

void print(li &l)
{
    li::iterator i = l.begin();
    while (i != l.end())
    {
        cout << *i << " " ;
        ++i;
    }
    cout << endl;
    return ;
}

int main(void)
{
    li l1;
```

```
    // Ajoute 5 à la liste :
    li::iterator i = l1.insert(l1.begin(), 5);
    print(l1);
    // Ajoute deux 3 à la liste :
    l1.insert(i, 2, 3);
    print(l1);
    // Insère le contenu de l1 dans une autre liste :
    li l2;
    l2.insert(l2.begin(), l1.begin(), l1.end());
    print(l2);
    return 0;
}
```

De manière similaire, il existe deux surcharges de la méthode `erase` qui permettent de spécifier de différentes manières les éléments qui doivent être supprimés d'une séquence. La première méthode prend en paramètre un itérateur sur l'élément à supprimer, et la deuxième un couple d'itérateurs donnant l'intervalle des éléments de la séquence qui doivent être supprimés. Ces deux méthodes retournent un itérateur sur l'élément suivant le dernier élément supprimé ou l'itérateur de fin de séquence s'il n'existe pas de tel élément. Par exemple, la suppression de tous les éléments d'une liste peut être réalisée de la manière suivante :

```
// Récupère un itérateur sur le premier
// élément de la liste :
list<int>::iterator i = instance.begin();
while (i != instance.end())
{
    i = instance.erase(i);
}
```

où `instance` est une instance de la séquence `Sequence`.

Vous noterez que la suppression d'un élément dans une séquence rend invalide tous les itérateurs sur cet élément. Il est à la charge du programmeur de s'assurer qu'il n'utilisera plus les itérateurs ainsi invalidés. La bibliothèque standard ne fournit aucun support pour le diagnostic de ce genre d'erreur.

> **Note :** En réalité, l'insertion d'un élément peut également invalider des itérateurs existants pour certaines séquences. Les effets de bord des méthodes d'insertion et de suppression des séquences seront détaillés pour chacune d'elle dans les sections qui leur sont dédiées.
>
> Il existe une méthode `clear` dont le rôle est de vider complètement un conteneur. On utilisera donc cette méthode dans la pratique, le code donné ci-dessous ne l'était qu'à titre d'exemple.

La complexité de toutes ces méthodes dépend directement du type de séquence sur lequel elles sont appliquées. Les avantages et les inconvénients de chaque séquence seront décrits dans la Section 17.2.2.

17.2.2. Les différents types de séquences

La bibliothèque standard fournit trois classes fondamentales de séquence. Ces trois classes sont respectivement la classe list, la classe vector et la classe deque. Chacune de ces classes possède ses spécificités en fonction desquelles le choix du programmeur devra se faire. De plus, la bibliothèque

standard fournit également des classes adaptatrices permettant de construire des conteneurs équivalents, mais disposant d'une interface plus standard et plus habituelle aux notions couramment utilisées en informatique. Toutes ces classes sont décrites dans cette section, les adaptateurs étant abordés en dernière partie.

17.2.2.1. Les listes

La classe `template` list est certainement l'une des plus importantes car, comme son nom l'indique, elle implémente une structure de liste chaînée d'éléments, ce qui est sans doute l'une des structures les plus utilisées en informatique. Cette structure est particulièrement adaptée pour les algorithmes qui parcourent les données dans un ordre séquentiel.

Les propriétés fondamentales des listes sont les suivantes :

- elles implémentent des itérateurs bidirectionnels. Cela signifie qu'il est facile de passer d'un élément au suivant ou au précédent, mais qu'il n'est pas possible d'accéder aux éléments de la liste de manière aléatoire ;
- elles permettent l'insertion et la suppression d'un élément avec un coût constant, et sans invalider les itérateurs ou les références sur les éléments de la liste existants. Dans le cas d'une suppression, seuls les itérateurs et les références sur les éléments supprimés sont invalidés.

Les listes offrent donc la plus grande souplesse possible sur les opérations d'insertion et de suppression des éléments, en contrepartie de quoi les accès sont restreints à un accès séquentiel.

Comme l'insertion et la suppression des éléments en tête et en queue de liste peuvent se faire sans recherche, ce sont évidemment les opérations les plus courantes. Par conséquent, la classe `template` list propose des méthodes spécifiques permettant de manipuler les éléments qui se trouvent en ces positions. L'insertion d'un élément peut donc être réalisée respectivement en tête et en queue de liste avec les méthodes `push_front` et `push_back`. Inversement, la suppression des éléments situés en ces emplacements est réalisée avec les méthodes `pop_front` et `pop_back`. Toutes ces méthodes ne renvoient aucune valeur, aussi l'accès aux deux éléments situés en tête et en queue de liste peut-il être réalisé respectivement par l'intermédiaire des accesseurs `front` et `back`, qui renvoient tous deux une référence (éventuellement constante si la liste est elle-même constante) sur ces éléments.

Exemple 17-3. Accès à la tête et à la queue d'une liste

```
#include <iostream>
#include <list>

using namespace std;

typedef list<int> li;

int main(void)
{
    li l1;
    l1.push_back(2);
    l1.push_back(5);
    cout << "Tête : " << l1.front() << endl;
    cout << "Queue : " << l1.back() << endl;
    l1.push_front(7);
    cout << "Tête : " << l1.front() << endl;
    cout << "Queue : " << l1.back() << endl;
```

```
    l1.pop_back();
    cout << "Tête : " << l1.front() << endl;
    cout << "Queue : " << l1.back() << endl;
    return 0;
}
```

Les listes disposent également de méthodes spécifiques qui permettent de leur appliquer des traitements qui leur sont propres. Ces méthodes sont décrites dans le tableau ci-dessous :

Tableau 17-1. Méthodes spécifiques aux listes

Méthode	Fonction
remove(const T &)	Permet d'éliminer tous les éléments d'une liste dont la valeur est égale à la valeur passée en paramètre. L'ordre relatif des éléments qui ne sont pas supprimés est inchangé. La complexité de cette méthode est linéaire en fonction du nombre d'éléments de la liste.
remove_if(Predicat)	Permet d'éliminer tous les éléments d'une liste qui vérifient le prédicat unaire passé en paramètre. L'ordre relatif des éléments qui ne sont pas supprimés est inchangé. La complexité de cette méthode est linéaire en fonction du nombre d'éléments de la liste.
unique(Predicat)	Permet d'éliminer tous les éléments pour lesquels le prédicat binaire passé en paramètre est vérifié avec comme valeur l'élément courant et son prédécesseur. Cette méthode permet d'éliminer les doublons successifs dans une liste selon un critère défini par le prédicat. Par souci de simplicité, il existe une surcharge de cette méthode qui ne prend pas de paramètres, et qui utilise un simple test d'égalité pour éliminer les doublons. L'ordre relatif des éléments qui ne sont pas supprimés est inchangé, et le nombre d'applications du prédicat est exactement le nombre d'éléments de la liste moins un si la liste n'est pas vide.
splice(iterator position, list<T, Allocator> liste, iterator premier, iterateur dernier)	Injecte le contenu de la liste fournie en deuxième paramètre dans la liste courante à partir de la position fournie en premier paramètre. Les éléments injectés sont les éléments de la liste source identifiés par les itérateurs `premier` et `dernier`. Ils sont supprimés de la liste source à la volée. Cette méthode dispose de deux autres surcharges, l'une ne fournissant pas d'itérateur de dernier élément et qui insère uniquement le premier élément, et l'autre ne fournissant aucun itérateur pour référencer les éléments à injecter. Cette dernière surcharge ne prend donc en paramètre que la position à laquelle les éléments doivent être insérés et la liste source elle-même. Dans ce cas, la totalité de la liste source est insérée en cet emplacement. Généralement, la complexité des méthodes `splice` est proportionnelle au nombre d'éléments injectés, sauf dans le cas de la dernière surcharge, qui s'exécute avec une complexité constante.
sort(Predicat)	Trie les éléments de la liste dans l'ordre défini par le prédicat binaire de comparaison passé en paramètre. Encore une fois, il existe une surcharge de cette méthode qui ne prend pas de paramètre et qui utilise l'opérateur d'infériorité pour comparer les éléments de la liste entre eux. L'ordre relatif des éléments équivalents (c'est-à-dire des éléments pour lesquels le prédicat de comparaison n'a pas pu statuer d'ordre bien défini) est inchangé à l'issue de l'opération de tri. On indique souvent cette propriété en disant que cette méthode est *stable*. La méthode `sort` s'applique avec une complexité égale à `N×ln(N)`, où `N` est le nombre d'éléments de la liste.

Méthode	Fonction
merge(list<T, Allocator>, Predicate)	Injecte les éléments de la liste fournie en premier paramètre dans la liste courante en conservant l'ordre défini par le prédicat binaire fourni en deuxième paramètre. Cette méthode suppose que la liste sur laquelle elle s'applique et la liste fournie en paramètre sont déjà triées selon ce prédicat, et garantit que la liste résultante sera toujours triée. La liste fournie en argument est vidée à l'issue de l'opération. Il existe également une surcharge de cette méthode qui ne prend pas de second paramètre et qui utilise l'opérateur d'infériorité pour comparer les éléments des deux listes. La complexité de cette méthode est proportionnelle à la somme des tailles des deux listes ainsi fusionnées.
reverse	Inverse l'ordre des éléments de la liste. Cette méthode s'exécute avec une complexité linéaire en fonction du nombre d'éléments de la liste.

Exemple 17-4. Manipulation de listes

```
#include <iostream>
#include <functional>
#include <list>

using namespace std;

typedef list<int> li;

void print(li &l)
{
    li::iterator i = l.begin();
    while (i != l.end())
    {
        cout << *i << " ";
        ++i;
    }
    cout << endl;
    return ;
}

bool parity_even(int i)
{
    return (i & 1) == 0;
}

int main(void)
{
    // Construit une liste exemple :
    li l;
    l.push_back(2);
    l.push_back(5);
    l.push_back(7);
    l.push_back(7);
    l.push_back(3);
    l.push_back(3);
    l.push_back(2);
    l.push_back(6);
    l.push_back(6);
```

```
    l.push_back(6);
    l.push_back(3);
    l.push_back(4);
    cout << "Liste de départ :" << endl;
    print(l);
    li l1;
    // Liste en ordre inverse :
    l1 = l;
    l1.reverse();
    cout << "Liste inverse :" << endl;
    print(l1);
    // Trie la liste :
    l1 = l;
    l1.sort();
    cout << "Liste triée : " << endl;
    print(l1);
    // Supprime tous les 3 :
    l1 = l;
    l1.remove(3);
    cout << "Liste sans 3 :" << endl;
    print(l1);
    // Supprime les doublons :
    l1 = l;
    l1.unique();
    cout << "Liste sans doublon :" << endl;
    print(l1);
    // Retire tous les nombres pairs :
    l1 = l;
    l1.remove_if(ptr_fun(&parity_even));
    cout << "Liste sans nombre pair :" << endl;
    print(l1);
    // Injecte une autre liste entre les 7 :
    l1 = l;
    li::iterator i = l1.begin();
    ++i; ++i; ++i;
    li l2;
    l2.push_back(35);
    l2.push_back(36);
    l2.push_back(37);
    l1.splice(i, l2, l2.begin(), l2.end());
    cout << "Fusion des deux listes :" << endl;
    print(l1);
    if (l2.size() == 0)
        cout << "l2 est vide" << endl;
    return 0;
}
```

17.2.2.2. Les vecteurs

La classe `template` vector de la bibliothèque standard fournit une structure de données dont la sémantique est proche de celle des tableaux de données classiques du langage C/C++. L'accès aux données de manière aléatoire est donc réalisable en un coût constant, mais l'insertion et la suppression des éléments dans un vecteur ont des conséquences nettement plus lourdes que dans le cas des listes.

Les propriétés des vecteurs sont les suivantes :

- les itérateurs permettent les accès aléatoires aux éléments du vecteur ;
- l'insertion ou la suppression d'un élément à la fin du vecteur se fait avec une complexité constante, mais l'insertion ou la suppression en tout autre point du vecteur se fait avec une complexité linéaire. Autrement dit, les opérations d'insertion ou de suppression nécessitent a priori de déplacer tous les éléments suivants, sauf si l'élément inséré ou supprimé se trouve en dernière position ;
- dans tous les cas, l'insertion d'un élément peut nécessiter une réallocation de mémoire. Cela a pour conséquence qu'en général, les données du vecteur peuvent être déplacées en mémoire et que les itérateurs et les références sur les éléments d'un vecteur sont a priori invalidés à la suite d'une insertion. Cependant, si aucune réallocation n'a lieu, les itérateurs et les références ne sont pas invalidés pour tous les éléments situés avant l'élément inséré ;
- la suppression d'un élément ne provoquant pas de réallocation, seuls les itérateurs et les références sur les éléments suivant l'élément supprimé sont invalidés.

> **Note :** Notez bien que les vecteurs peuvent effectuer une réallocation même lorsque l'insertion se fait en dernière position. Dans ce cas, le coût de l'insertion est bien entendu très élevé. Toutefois, l'algorithme de réallocation utilisé est suffisament évolué pour garantir que ce coût est constant en moyenne (donc de complexité constante). Autrement dit, les réallocations ne se font que très rarement.

Tout comme la classe list, la classe `template` vector dispose de méthodes `front` et `back` qui permettent d'accéder respectivement au premier et au dernier élément des vecteurs. Cependant, contrairement aux listes, seule les méthodes `push_back` et `pop_back` sont définies, car les vecteurs ne permettent pas d'insérer et de supprimer leurs premiers éléments de manière rapide.

En revanche, comme nous l'avons déjà dit, les vecteurs ont la même sémantique que les tableaux et permettent donc un accès rapide à tous leurs éléments. La classe vector définit donc une méthode `at` qui prend en paramètre l'indice d'un élément dans le vecteur et qui renvoie une référence, éventuellement constante si le vecteur l'est lui-même, sur cet élément. Si l'indice fourni en paramètre référence un élément situé en dehors du vecteur, la méthode `at` lance une exception out_of_range. De même, il est possible d'appliquer l'opérateur `[]` utilisé habituellement pour accéder aux éléments des tableaux. Cet opérateur se comporte exactement comme la méthode `at`, et est donc susceptible de lancer une exception out_of_range.

Exemple 17-5. Accès aux éléments d'un vecteur

```
#include <iostream>
#include <vector>

using namespace std;

int main(void)
{
    typedef vector<int> vi;
    // Crée un vecteur de 10 éléments :
    vi v(10);
    // Modifie quelques éléments :
    v.at(2) = 2;
    v.at(5) = 7;
    // Redimensionne le vecteur :
    v.resize(11);
    v.at(10) = 5;
```

```
    // Ajoute un élément à la fin du vecteur :
    v.push_back(13);
    // Affiche le vecteur en utilisant l'opérateur [] :
    for (int i=0; i<v.size(); ++i)
    {
        cout << v[i] << endl;
    }
    return 0;
}
```

Par ailleurs, la bibliothèque standard définit une spécialisation de la classe `template` vector pour le type bool. Cette spécialisation a essentiellement pour but de réduire la consommation mémoire des vecteurs de booléens, en codant ceux-ci à raison d'un bit par booléen seulement. Les références des éléments des vecteurs de booléens ne sont donc pas réellement des booléens, mais plutôt une classe spéciale qui simule ces booléens tout en manipulant les bits réellement stockés dans ces vecteurs. Ce mécanisme est donc complètement transparent pour l'utilisateur, et les vecteurs de booléens se manipulent exactement comme les vecteurs classiques.

> **Note :** La classe de référence des vecteurs de booléens disposent toutefois d'une méthode `flip` dont le rôle est d'inverser la valeur du bit correspondant au booléen que la référence représente. Cette méthode peut être pratique à utiliser lorsqu'on désire inverser rapidement la valeur d'un des éléments du vecteur.

17.2.2.3. Les deques

Pour ceux à qui les listes et les vecteurs ne conviennent pas, la bibliothèque standard fournit un conteneur plus évolué qui offre un autre compromis entre la rapidité d'accès aux éléments et la souplesse dans les opérations d'ajout ou de suppression. Il s'agit de la classe `template` deque, qui implémente une forme de tampon circulaire dynamique.

Les propriétés des deques sont les suivantes :

- les itérateurs des deques permettent les accès aléatoires à leurs éléments ;
- l'insertion et la suppression des éléments en première et en dernière position se fait avec un coût constant. Notez ici que ce coût est toujours le même, et que, contrairement aux vecteurs, il ne s'agit pas d'un coût amorti (autrement dit, ce n'est pas une moyenne). En revanche, tout comme pour les vecteurs, l'insertion et la suppression aux autres positions se fait avec une complexité linéaire ;
- contrairement aux vecteurs, tous les itérateurs et toutes les références sur les éléments de la deque deviennent systématiquement invalides lors d'une insertion ou d'une suppression d'élément aux autres positions que la première et la dernière ;
- de même, l'insertion d'un élément en première et dernière position invalide tous les itérateurs sur les éléments de la deque. En revanche, les références sur les éléments restent valides. Remarquez que la suppression d'un élément en première et en dernière position n'a aucun impact sur les itérateurs et les références des éléments autres que ceux qui sont supprimés.

Comme vous pouvez le constater, les deques sont donc extrêmement bien adaptés aux opérations d'insertion et de suppression en première et en dernière position, tout en fournissant un accès rapide à leurs éléments. En revanche, les itérateurs existants sont systématiquement invalidés, quel que soit le type d'opération effectuée, hormis la suppression en tête et en fin de deque.

Comme elle permet un accès rapide à tous ses éléments, la classe `template` deque dispose de toutes les méthodes d'insertion et de suppression d'éléments des listes et des vecteurs. Outre les méthodes `push_front`, `pop_front`, `push_back`, `pop_back` et les accesseurs `front` et `back`, la classe deque définit donc la méthode `at`, ainsi que l'opérateur d'accès aux éléments de tableaux `[]`. L'utilisation de ces méthodes est strictement identique à celle des méthodes homonymes des classes `list` et `vector` et ne devrait donc pas poser de problème particulier.

17.2.2.4. Les adaptateurs de séquences

Les classes des séquences de base list, vector et deque sont supposées satisfaire à la plupart des besoins courants des programmeurs. Cependant, la bibliothèque standard fournit des adaptateurs pour transformer ces classes en d'autres structures de données plus classiques. Ces adaptateurs permettent de construire des piles, des files et des files de priorité.

17.2.2.4.1. Les piles

Les piles sont des structures de données qui se comportent, comme leur nom l'indique, comme un empilement d'objets. Elles ne permettent donc d'accéder qu'aux éléments situés en haut de la pile, et la récupération des éléments se fait dans l'ordre inverse de leur empilement. En raison de cette propriété, on les appelle également couramment *LIFO*, acronyme de l'anglais « Last In First Out » (dernier entré, premier sorti).

La classe adaptatrice définie par la bibliothèque standard C++ pour implémenter les piles est la classe `template` stack. Cette classe utilise deux paramètres `template` : le type des données lui-même et le type d'une classe de séquence implémentant au moins les méthodes `back`, `push_back` et `pop_back`. Il est donc parfaitement possible d'utiliser les listes, deques et vecteurs pour implémenter une pile à l'aide de cet adaptateur. Par défaut, la classe stack utilise une deque, et il n'est donc généralement pas nécessaire de spécifier le type du conteneur à utiliser pour réaliser la pile.

L'interface des piles se réduit au strict minimum, puisqu'elles ne permettent de manipuler que leur sommet. La méthode `push` permet d'empiler un élément sur la pile, et la méthode `pop` de l'en retirer. Ces deux méthodes ne renvoient rien, l'accès à l'élément situé au sommet de la pile se fait donc par l'intermédiaire de la méthode `top`.

Exemple 17-6. Utilisation d'une pile

```
#include <iostream>
#include <stack>

using namespace std;

int main(void)
{
    typedef stack<int> si;
    // Crée une pile :
    si s;
    // Empile quelques éléments :
    s.push(2);
    s.push(5);
    s.push(8);
    // Affiche les éléments en ordre inverse :
    while (!s.empty())
    {
        cout << s.top() << endl;
```

```
        s.pop();
    }
    return 0;
}
```

17.2.2.4.2. Les files

Les files sont des structures de données similaires aux piles, à la différence près que les éléments sont mis les uns à la suite des autres au lieu d'être empilés. Leur comportement est donc celui d'une file d'attente où tout le monde serait honnête (c'est-à-dire que personne ne doublerait les autres). Les derniers entrés sont donc ceux qui sortent également en dernier, d'où leur dénomination de *FIFO* (de l'anglais « First In First Out »).

Les files sont implémentées par la classe `template` queue. Cette classe utilise comme paramètre `template` le type des éléments stockés ainsi que le type d'un conteneur de type séquence pour lequel les méthodes `front`, `back`, `push_back` et `pop_front` sont implémentées. En pratique, il est possible d'utiliser les listes et les deques, la classe queue utilisant d'ailleurs ce type de séquence par défaut comme conteneur sous-jacent.

> **Note :** Ne confondez pas la classe queue et la classe deque. La première n'est qu'un simple adaptateur pour les files d'éléments, alors que la deuxième est un conteneur très évolué et beaucoup plus complexe.

Les méthodes fournies par les files sont les méthodes `front` et `back`, qui permettent d'accéder respectivement au premier et au dernier élément de la file d'attente, ainsi que les méthodes `push` et `pop`, qui permettent respectivement d'ajouter un élément à la fin de la file et de supprimer l'élément qui se trouve en tête de file.

Exemple 17-7. Utilisation d'une file

```
#include <iostream>
#include <queue>

using namespace std;

int main(void)
{
    typedef queue<int> qi;
    // Crée une file :
    qi q;
    // Ajoute quelques éléments :
    q.push(2);
    q.push(5);
    q.push(8);
    // Affiche récupère et affiche les éléments :
    while (!q.empty())
    {
        cout << q.front() << endl;
        q.pop();
    }
    return 0;
}
```

17.2.2.4.3. Les files de priorités

Enfin, la bibliothèque standard fournit un adaptateur permettant d'implémenter les files de priorités. Les files de priorités ressemblent aux files classiques, mais ne fonctionnent pas de la même manière. En effet, contrairement aux files normales, l'élément qui se trouve en première position n'est pas toujours le premier élément qui a été placé dans la file, mais celui qui dispose de la plus grande valeur. C'est cette propriété qui a donné son nom aux files de priorités, car la priorité d'un élément est ici donnée par sa valeur. Bien entendu, la bibliothèque standard permet à l'utilisateur de définir son propre opérateur de comparaison, afin de lui laisser spécifier l'ordre qu'il veut utiliser pour définir la priorité des éléments.

> **Note :** On prendra garde au fait que la bibliothèque standard n'impose pas aux files de priorités de se comporter comme des files classiques avec les éléments de priorités égales. Cela signifie que si plusieurs éléments de priorité égale sont insérés dans une file de priorité, ils n'en sortiront pas forcément dans l'ordre d'insertion. On dit généralement que les algorithmes utilisés par les files de priorités ne sont pas *stables* pour traduire cette propriété.

La classe `template` fournie par la bibliothèque standard pour faciliter l'implémentation des files de priorité est la classe priority_queue. Cette classe prend trois paramètres `template` : le type des éléments stockés, le type d'un conteneur de type séquence permettant un accès direct à ses éléments et implémentant les méthodes `front`, `push_back` et `pop_back`, et le type d'un prédicat binaire à utiliser pour la comparaison des priorités des éléments. On peut donc implémenter une file de priorité à partir d'un vecteur ou d'une deque, sachant que, par défaut, la classe priority_queue utilise un vecteur. Le prédicat de comparaison utilisé par défaut est le foncteur less<T>, qui effectue une comparaison à l'aide de l'opérateur d'infériorité des éléments stockés dans la file.

Comme les files de priorités se réorganisent à chaque fois qu'un nouvel élément est ajouté en fin de file, et que cet élément ne se retrouve par conséquent pas forcément en dernière position s'il est de priorité élevée, accéder au dernier élément des files de priorité n'a pas de sens. Il n'existe donc qu'une seule méthode permettant d'accéder à l'élément le plus important de la pile : la méthode `top`. En revanche, les files de priorité implémentent effectivement les méthodes `push` et `pop`, qui permettent respectivement d'ajouter un élément dans la file de priorité et de supprimer l'élément le plus important de cette file.

Exemple 17-8. Utilisation d'une file de priorité

```
#include <iostream>
#include <queue>

using namespace std;

// Type des données stockées dans la file :
struct A
{
    int k;         // Priorité
    const char *t; // Valeur
    A() : k(0), t(0) {}
    A(int k, const char *t) : k(k), t(t) {}
};

// Foncteur de comparaison selon les priorités :
class C
{
public:
```

```
    bool operator()(const A &a1, const A &a2)
    {
        return a1.k < a2.k ;
    }
};

int main(void)
{
    // Construit quelques objets :
    A a1(1, "Priorité faible");
    A a2(2, "Priorité moyenne 1");
    A a3(2, "Priorité moyenne 2");
    A a4(3, "Priorité haute 1");
    A a5(3, "Priorité haute 2");
    // Construit une file de priorité :
    priority_queue<A, vector<A>, C> pq;
    // Ajoute les éléments :
    pq.push(a5);
    pq.push(a3);
    pq.push(a1);
    pq.push(a2);
    pq.push(a4);
    // Récupère les éléments par ordre de priorité :
    while (!pq.empty())
    {
        cout << pq.top().t << endl;
        pq.pop();
    }
    return 0;
}
```

Note : En raison de la nécessité de réorganiser l'ordre du conteneur sous-jacent à chaque ajout ou suppression d'un élément, les méthodes `push` et `pop` s'exécutent avec une complexité en `ln(N)`, où `N` est le nombre d'éléments présents dans la file de priorité.

Les files de priorité utilisent en interne la structure de tas, que l'on décrira dans le chapitre traitant des algorithmes de la bibliothèque standard à la section Section 18.3.1.

17.3. Les conteneurs associatifs

Contrairement aux séquences, les conteneurs associatifs sont capables d'identifier leurs éléments à l'aide de la valeur de leur clef. Grâce à ces clefs, les conteneurs associatifs sont capables d'effectuer des recherches d'éléments de manière extrêmement performante. En effet, les opérations de recherche se font généralement avec un coût logarithmique seulement, ce qui reste généralement raisonnable même lorsque le nombre d'éléments stockés devient grand. Les conteneurs associatifs sont donc particulièrement adaptés lorsqu'on a besoin de réaliser un grand nombre d'opération de recherche.

La bibliothèque standard distingue deux types de conteneurs associatifs : les conteneurs qui différencient la valeur de la clef de la valeur de l'objet lui-même et les conteneurs qui considèrent que les objets sont leur propre clef. Les conteneurs de la première catégorie constituent ce que l'on appelle

des *associations* car ils permettent d'associer des clefs aux valeurs des objets. Les conteneurs associatifs de la deuxième catégorie sont appelés quant à eux des *ensembles*, en raison du fait qu'ils servent généralement à indiquer si un objet fait partie ou non d'un ensemble d'objets. On ne s'intéresse dans ce cas pas à la valeur de l'objet, puisqu'on la connaît déjà si on dispose de sa clef, mais plutôt à son appartenance ou non à un ensemble donné.

Si tous les conteneurs associatifs utilisent la notion de clef, tous ne se comportent pas de manière identique quant à l'utilisation qu'ils en font. Pour certains conteneurs, que l'on qualifie de conteneurs « *à clefs uniques* », chaque élément contenu doit avoir une clef qui lui est propre. Il est donc impossible d'insérer plusieurs éléments distincts avec la même clef dans ces conteneurs. En revanche, les conteneurs associatif dits « *à clefs multiples* » permettent l'utilisation d'une même valeur de clef pour plusieurs objets distincts. L'opération de recherche d'un objet à partir de sa clef peut donc, dans ce cas, renvoyer plus d'un seul objet.

La bibliothèque standard fournit donc quatre types de conteneurs au total, selon que ce sont des associations ou des ensembles, et selon que ce sont des conteneurs associatifs à clefs multiples ou non. Les associations à clefs uniques et à clefs multiple sont implémentées respectivement par les classes `template` map et multimap, et les ensembles à clefs uniques et à clefs multiples par les classes `template` set et multiset. Cependant, bien que ces classes se comportent de manière profondément différentes, elles fournissent les mêmes méthodes permettant de les manipuler. Les conteneurs associatifs sont donc moins hétéroclites que les séquences, et leur manipulation en est de beaucoup facilitée.

Les sections suivantes présentent les différentes fonctionnalités des conteneurs associatifs dans leur ensemble. Les exemples seront donnés en utilisant la plupart du temps la classe `template` map, car c'est certainement la classe la plus utilisée en pratique en raison de sa capacité à stocker et à retrouver rapidement des objets identifiés de manière unique par un identifiant. Cependant, certains exemples utiliseront des conteneurs à clefs multiples afin de bien montrer les rares différences qui existent entre les conteneurs à clefs uniques et les conteneurs à clefs multiples.

17.3.1. Généralités et propriétés de base des clefs

La contrainte fondamentale que les algorithmes des conteneurs associatifs imposent est qu'il existe une relation d'ordre pour le type de donnée utilisé pour les clefs des objets. Cette relation peut être définie soit implicitement par un opérateur d'infériorité, soit par un foncteur que l'on peut spécifier en tant que paramètre `template` des classes des conteneurs.

Alors que l'ordre de la suite des éléments stockés dans les séquences est très important, ce n'est pas le cas avec les conteneurs associatifs, car ceux-ci se basent exclusivement sur l'ordre des clefs des objets. En revanche, la bibliothèque standard C++ garantit que le sens de parcours utilisé par les itérateurs des conteneurs associatifs est non décroissant sur les clefs des objets itérés. Cela signifie que le test d'infériorité strict entre la clef de l'élément suivant et la clef de l'élément courant est toujours faux, ou, autrement dit, l'élément suivant n'est pas plus petit que l'élément courant.

> **Note :** Attention, cela ne signifie aucunement que les éléments sont classés dans l'ordre croissant des clefs. En effet, l'existence d'un opérateur d'infériorité n'implique pas forcément celle d'un opérateur de supériorité d'une part, et deux valeurs comparables par cet opérateur ne le sont pas forcément par l'opérateur de supériorité. L'élément suivant n'est donc pas forcément plus grand que l'élément courant. En particulier, pour les conteneurs à clefs multiples, les clefs de deux éléments successifs peuvent être égales.
>
> En revanche, le classement utilisé par les itérateurs des conteneurs à clefs uniques est plus fort, puisque dans ce cas, on n'a pas à se soucier des clefs ayant la même valeur. La séquence des valeurs itérées est donc cette fois strictement croissante, c'est-à-dire que la clef de l'élément courant est toujours strictement inférieure à la clef de l'élément suivant.

Comme pour tous les conteneurs, le type des éléments stockés par les conteneurs associatifs est le type value_type. Cependant, contrairement aux séquences, ce type n'est pas toujours le type `template` par lequel le conteneur est paramétré. En effet, ce type est une paire contenant le couple de valeurs formé par la clef et par l'objet lui-même pour toutes les associations (c'est-à-dire pour les map et les multimap). Dans ce cas, les méthodes du conteneur qui doivent effectuer des comparaisons sur les objets se basent uniquement sur le champ *`first`* de la paire encapsulant le couple (clef, valeur) de chaque objet. Autrement dit, les comparaisons d'objets sont toujours définies sur les clefs, et jamais sur les objets eux-mêmes. Bien entendu, pour les ensembles, le type value_type est strictement équivalent au type `template` par lequel ils sont paramétrés.

Pour simplifier l'utilisation de leurs clefs, les conteneurs associatifs définissent quelques types complémentaires de ceux que l'on a déjà présentés dans la Section 17.1.2. Le plus important de ces types est sans doute le type key_type qui, comme son nom l'indique, représente le type des clefs utilisées par ce conteneur. Ce type constitue donc, avec le type value_type, l'essentiel des informations de typage des conteneurs associatifs. Enfin, les conteneurs définissent également des types de prédicats permettant d'effectuer des comparaisons entre deux clefs et entre deux objets de type value_type. Il s'agit des types key_compare et value_compare.

17.3.2. Construction et initialisation

Les conteneurs associatifs disposent de plusieurs surcharges de leurs constructeurs qui permettent de les créer et de les initialiser directement. De manière générale, ces constructeurs prennent tous deux paramètres afin de laisser au programmeur la possibilité de définir la valeur du foncteur qu'ils doivent utiliser pour comparer les clefs, ainsi qu'une instance de l'allocateur à utiliser pour les opérations mémoire. Comme pour les séquences, ces paramètres disposent de valeurs par défaut, si bien qu'en général il n'est pas nécessaire de les préciser.

Hormis le constructeur de copie et le constructeur par défaut, les conteneurs associatifs fournissent un troisième constructeur permettant de les initialiser à partir d'une série d'objets. Ces objets sont spécifiés par deux itérateurs, le premier indiquant le premier objet à insérer dans le conteneur et le deuxième l'itérateur référençant l'élément suivant le dernier élément à insérer. L'utilisation de ce constructeur est semblable au constructeur du même type défini pour les séquences et ne devrait donc pas poser de problème particulier.

Exemple 17-9. Construction et initialisation d'une association simple

```
#include <iostream>
#include <map>
#include <list>

using namespace std;

int main(void)
{
    typedef map<int, char *> Int2String;
    // Remplit une liste d'éléments pour ces maps :
    typedef list<pair<int, char *> > lv;
    lv l;
    l.push_back(lv::value_type(1, "Un"));
    l.push_back(lv::value_type(2, "Deux"));
    l.push_back(lv::value_type(5, "Trois"));
    l.push_back(lv::value_type(6, "Quatre"));
```

```
    // Construit une map et l'initialise avec la liste :
    Int2String i2s(l.begin(), l.end());
    // Affiche le contenu de la map :
    Int2String::iterator i = i2s.begin();
    while (i != i2s.end())
    {
        cout << i->second << endl;
        ++i;
    }
    return 0;
}
```

Note : Contrairement aux séquences, les conteneurs associatifs ne disposent pas de méthode `assign` permettant d'initialiser un conteneur avec des objets provenant d'une séquence ou d'un autre conteneur associatif. En revanche, ils disposent d'un constructeur et d'un opérateur de copie.

17.3.3. Ajout et suppression d'éléments

Du fait de l'existence des clefs, les méthodes d'insertion et de suppression des conteneurs associatifs sont légèrement différentes de celles des séquences. De plus, elles n'ont pas tout à fait la même signification. En effet, les méthodes d'insertion des conteneurs associatifs ne permettent pas, contrairement à celles des séquences, de spécifier l'emplacement où un élément doit être inséré puisque l'ordre des éléments est imposé par la valeur de leur clef. Les méthodes d'insertion des conteneurs associatifs sont présentées ci-dessous :

`iterator insert(iterator i, const value_type &valeur)`

Insère la valeur `valeur` dans le conteneur. L'itérateur `i` indique l'emplacement probable dans le conteneur où l'insertion doit être faite. Cette méthode peut donc être utilisée pour les algorithmes qui connaissent déjà plus ou moins l'ordre des éléments qu'ils insèrent dans le conteneur afin d'optimiser les performances du programme. En général, l'insertion se fait avec une complexité de `ln(N)` (où `N` est le nombre d'éléments déjà présents dans le conteneur). Toutefois, si l'élément est inséré après l'itérateur `i` dans le conteneur, la complexité est constante. L'insertion se fait systématiquement pour les conteneurs à clefs multiples, mais peut ne pas avoir lieu si un élément de même clef que celui que l'on veut insérer est déjà présent pour les conteneurs à clefs uniques. Dans tous les cas, la valeur retournée est un itérateur référençant l'élément inséré ou l'élément ayant la même clef que l'élément à insérer.

`void insert(iterator premier, iterator dernier)`

Insère les éléments de l'intervalle défini par les itérateurs `premier` et `dernier` dans le conteneur. La complexité de cette méthode est `n×ln(n+N)` en général, où `N` est le nombre d'éléments déjà présents dans le conteneur et `n` est le nombre d'éléments à insérer. Toutefois, si les éléments à insérer sont classés dans l'ordre de l'opérateur de comparaison utilisé par le conteneur, l'insertion se fait avec un coût proportionnel au nombre d'éléments à insérer.

`pair<iterator, bool> insert(const value_type &valeur)`

Insère ou tente d'insérer un nouvel élément dans un conteneur à clefs uniques. Cette méthode renvoie une paire contenant l'itérateur référençant cet élément dans le conteneur et un booléen

indiquant si l'insertion a effectivement eu lieu. Cette méthode n'est définie que pour les conteneurs associatifs à clefs uniques (c'est-à-dire les map et les set). Si aucun élément du conteneur ne correspond à la clef de l'élément passé en paramètre, cet élément est inséré dans le conteneur et la valeur renvoyée dans le deuxième champ de la paire vaut `true`. En revanche, si un autre élément utilisant cette clef existe déjà dans le conteneur, aucune insertion n'a lieu et le deuxième champ de la paire renvoyée vaut alors `false`. Dans tous les cas, l'itérateur stocké dans le premier champ de la valeur de retour référence l'élément inséré ou trouvé dans le conteneur. La complexité de cette méthode est logarithmique.

`iterator insert(const value_type &valeur)`

Insère un nouvel élément dans un conteneur à clefs multiples. Cette insertion se produit qu'il y ait déjà ou non un autre élément utilisant la même clef dans le conteneur. La valeur retournée est un itérateur référençant le nouvel élément inséré. Vous ne trouverez cette méthode que sur les conteneurs associatifs à clefs multiples, c'est-a-dire sur les multimap et les multiset. La complexité de cette méthode est logarithmique.

Comme pour les séquences, la suppression des éléments des conteneurs associatifs se fait à l'aide des surcharges de la méthode `erase`. Les différentes versions de cette méthode sont indiquées ci-dessous :

`void erase(iterator i)`

Permet de supprimer l'élément référencé par l'itérateur `i`. Cette opération a un coût amorti constant car aucune recherche n'est nécessaire pour localiser l'élément.

`void erase(iterator premier, iterator dernier)`

Supprime tous les éléments de l'intervalle défini par les deux itérateurs `premier` et `dernier`. La complexité de cette opération est `ln(N)+n`, où `N` est le nombre d'éléments du conteneur avant suppression et `n` est le nombre d'éléments qui seront supprimés.

`size_type erase(key_type clef)`

Supprime tous les éléments dont la clef est égale à la valeur passée en paramètre. Cette opération a pour complexité `ln(N)+n`, où `N` est le nombre d'éléments du conteneur avant suppression et `n` est le nombre d'éléments qui seront supprimés. Cette fonction retourne le nombre d'éléments effectivement supprimés. Ce nombre peut être nul si aucun élément ne correspond à la clef fournie en paramètre, ou valoir 1 pour les conteneurs à clefs uniques, ou être supérieur à 1 pour les conteneurs à clefs multiples.

Les conteneurs associatifs disposent également, tout comme les séquences, d'une méthode `clear` permettant de vider complètement un conteneur. Cette opération est réalisée avec un coût proportionnel au nombre d'éléments se trouvant dans le conteneur.

Exemple 17-10. Insertion et suppression d'éléments d'une association

```
#include <iostream>
#include <map>

using namespace std;

typedef map<int, char *> Int2String;

void print(Int2String &m)
```

```
{
    Int2String::iterator i = m.begin();
    while (i != m.end())
    {
        cout << i->second << endl;
        ++i;
    }
    return ;
}

int main(void)
{
    // Construit une association Entier -> Chaîne :
    Int2String m;
    // Ajoute quelques éléments :
    m.insert(Int2String::value_type(2, "Deux"));
    pair<Int2String::iterator, bool> res =
        m.insert(Int2String::value_type(3, "Trois"));
    // On peut aussi spécifier un indice sur
    // l'emplacement où l'insertion aura lieu :
    m.insert(res.first,
        Int2String::value_type(5, "Cinq"));
    // Affiche le contenu de l'association :
    print(m);
    // Supprime l'élément de clef 2 :
    m.erase(2);
    // Supprime l'élément "Trois" par son itérateur :
    m.erase(res.first);
    print(m);
    return 0;
}
```

17.3.4. Fonctions de recherche

Les fonctions de recherche des conteneurs associatifs sont puissantes et nombreuses. Ces méthodes sont décrites ci-dessous :

`iterator find(key_type clef)`

Renvoie un itérateur référençant un élément du conteneur dont la clef est égale à la valeur passée en paramètre. Dans le cas des conteneurs à clefs multiples, l'itérateur renvoyé référence un des éléments dont la clef est égale à la valeur passée en paramètre. Attention, ce n'est pas forcément le premier élément du conteneur vérifiant cette propriété. Si aucun élément ne correspond à la clef, l'itérateur de fin du conteneur est renvoyé.

`iterator lower_bound(key_type clef)`

Renvoie un itérateur sur le premier élément du conteneur dont la clef est égale à la valeur passée en paramètre. Les valeurs suivantes de l'itérateur référenceront les éléments suivants dont la clef est supérieure ou égale à la clef de cet élément.

`iterator upper_bound(key_type clef)`

Renvoie un itérateur sur l'élément suivant le dernier élément dont la clef est égale à la valeur passée en paramètre. S'il n'y a pas de tel élément, c'est-à-dire si le dernier élément du conteneur utilise cette valeur de clef, renvoie l'itérateur de fin du conteneur.

`pair<iterator, iterator> equal_range(key_type clef)`

Renvoie une paire d'itérateurs égaux respectivement aux itérateurs renvoyés par les méthodes `lower_bound` et `upper_bound`. Cette paire d'itérateurs référence donc tous les éléments du conteneur dont la clef est égale à la valeur passée en paramètre.

Exemple 17-11. Recherche dans une association

```
#include <iostream>
#include <map>

using namespace std;

int main(void)
{
    // Déclare une map à clefs multiples :
    typedef multimap<int, char *> Int2String;
    Int2String m;
    // Remplit la map :
    m.insert(Int2String::value_type(2, "Deux"));
    m.insert(Int2String::value_type(3, "Drei"));
    m.insert(Int2String::value_type(1, "Un"));
    m.insert(Int2String::value_type(3, "Three"));
    m.insert(Int2String::value_type(4, "Quatre"));
    m.insert(Int2String::value_type(3, "Trois"));
    // Recherche un élément de clef 4 et l'affiche :
    Int2String::iterator i = m.find(4);
    cout << i->first << " : " << i->second << endl;
    // Recherche le premier élément de clef 3 :
    i = m.lower_bound(3);
    // Affiche tous les éléments dont la clef vaut 3 :
    while (i != m.upper_bound(3))
    {
        cout << i->first << " : " << i->second << endl;
        ++i;
    }
    // Effectue la même opération, mais de manière plus efficace
    // (upper_bound n'est pas appelée à chaque itération) :
    pair<Int2String::iterator, Int2String::iterator> p =
        m.equal_range(3);
    for (i = p.first; i != p.second; ++i)
    {
        cout << i->first << " : " << i->second << endl;
    }
    return 0;
}
```

Note : Il existe également des surcharges `const` pour ces quatre méthodes de recherche afin de pouvoir les utiliser sur des conteneurs constants. Ces méthodes retournent des valeurs de type

const_iterator au lieu des itérateurs classiques, car il est interdit de modifier les valeurs stockées dans un conteneur de type `const`.

La classe `template` map fournit également une surcharge pour l'opérateur d'accès aux membres de tableau `[]`. Cet opérateur renvoie la valeur de l'élément référencé par sa clef et permet d'obtenir directement cette valeur sans passer par la méthode `find` et un déréférencement de l'itérateur ainsi obtenu. Cet opérateur insère automatiquement un nouvel élément construit avec la valeur par défaut du type des éléments stockés dans la map si aucun élément ne correspond à la clef fournie en paramètre. Contrairement à l'opérateur `[]` des classes vector et deque, cet opérateur ne renvoie donc jamais l'exception out_of_range.

Les recherches dans les conteneurs associatifs s'appuient sur le fait que les objets disposent d'une relation d'ordre induite par le foncteur less appliqué sur le type des données qu'ils manipulent. Ce comportement est généralement celui qui est souhaité, mais il existe des situations où ce foncteur ne convient pas. Par exemple, on peut désirer que le classement des objets se fasse sur une de leur donnée membre seulement, ou que la fonction de comparaison utilisée pour classer les objets soit différente de celle induite par le foncteur less. La bibliothèque standard fournit donc la possibilité de spécifier un foncteur de comparaison pour chaque conteneur associatif, en tant que paramètre template complémentaire au type de données des objets contenus. Ce foncteur doit, s'il est spécifié, être précisé avant le type de l'allocateur mémoire à utiliser. Il pourra être construit à partir des facilités fournies par la bibliothèque standard pour la création et la manipulation des foncteurs.

Exemple 17-12. Utilisation d'un foncteur de comparaison personnalisé

```
#include <iostream>
#include <map>
#include <string>
#include <functional>
#include <cstring>

using namespace std;

// Fonction de comparaison de chaînes de caractères
// non sensible à la casse des lettres :
bool stringless_nocase(const string &s1, const string &s2)
{
    return (strcasecmp(s1.c_str(), s2.c_str()) < 0);
}

int main(void)
{
    // Définit le type des associations chaînes -> entiers
    // dont la clef est indexée sans tenir compte
    // de la casse des lettres :
    typedef map<string, int,
        pointer_to_binary_function<const string &,
            const string &, bool> > String2Int;
    String2Int m(ptr_fun(stringless_nocase));
    // Insère quelques éléments dans la map :
    m.insert(String2Int::value_type("a. Un", 1));
    m.insert(String2Int::value_type("B. Deux", 2));
    m.insert(String2Int::value_type("c. Trois", 3));
    // Affiche le contenu de la map :
    String2Int::iterator i = m.begin();
    while (i != m.end())
```

```
    {
        cout << i->first << " : " << i->second << endl;
        ++i;
    }
    return 0;
}
```

Dans cet exemple, le type du foncteur est spécifié en troisième paramètre de la classe `template` map. Ce type est une instance de la classe `template` pointer_to_binary_function pour les types string et bool. Comme on l'a vu dans la Section 13.5, cette classe permet d'encapsuler toute fonction binaire dans un foncteur binaire. Il ne reste donc qu'à spécifier l'instance du foncteur que la classe `template` map doit utiliser, en la lui fournissant dans son constructeur. L'exemple précédent utilise la fonction utilitaire `ptr_fun` de la bibliothèque standard pour construire ce foncteur à partir de la fonction `stringless_nocase`.

En fait, il est possible de passer des foncteurs beaucoup plus évolués à la classe map, qui peuvent éventuellement être paramétrés par d'autres paramètres que la fonction de comparaison à utiliser pour comparer deux clefs. Cependant, il est rare d'avoir à écrire de tels foncteurs et même, en général, il est courant que la fonction binaire utilisée soit toujours la même. Dans ce cas, il est plus simple de définir directement le foncteur et de laisser le constructeur de la classe map prendre sa valeur par défaut. Ainsi, seul le paramètre `template` donnant le type du foncteur doit être spécifié, et l'utilisation des conteneurs associatif en est d'autant facilitée. L'exemple suivant montre comment la comparaison de chaînes de caractères non sensible à la casse peut être implémentée de manière simplifiée.

Exemple 17-13. Définition directe du foncteur de comparaison pour les recherches

```
#include <iostream>
#include <string>
#include <map>
#include <functional>
#include <cstring>

using namespace std;

// Classe de comparaison de chaînes de caractères :
class StringLessNoCase : public binary_function<string, string, bool>
{
public:
    bool operator()(const string &s1, const string &s2)
    {
        return (strcasecmp(s1.c_str(), s2.c_str()) < 0);
    }
};

int main(void)
{
    // Définition du type des associations chaînes -> entiers
    // en spécifiant directement le type de foncteur à utiliser
    // pour les comparaisons de clefs :
    typedef map<string, int, StringLessNoCase> String2Int;
    // Instanciation d'une association en utilisant
    // la valeur par défaut du foncteur de comparaison :
    String2Int m;
    // Utilisation de la map :
    m.insert(String2Int::value_type("a. Un", 1));
    m.insert(String2Int::value_type("B. Deux", 2));
```

```
    m.insert(String2Int::value_type("c. Trois", 3));
    String2Int::iterator i = m.begin();
    while (i != m.end())
    {
        cout << i->first << " : " << i->second << endl;
        ++i;
    }
    return 0;
}
```

Note : Les deux exemples précédents utilisent la fonction `strcasecmp` de la bibliothèque C standard pour effectuer des comparaisons de chaînes qui ne tiennent pas compte de la casse des caractères. Cette fonction s'utilise comme la fonction `strcmp`, qui compare deux chaînes et renvoie un entier dont le signe indique si la première chaîne est plus petite ou plus grande que la deuxième. Ces fonctions renvoient 0 si les deux chaînes sont strictement égales. Si vous désirez en savoir plus sur les fonctions de manipulation de chaînes de la bibliothèque C, veuillez vous référer à la bibliographie.

Pour finir, sachez que les conteneurs associatifs disposent d'une méthode `count` qui renvoie le nombre d'éléments du conteneur dont la clef est égale à la valeur passée en premier paramètre. Cette méthode retourne donc une valeur du type size_type du conteneur, valeur qui peut valoir 0 ou 1 pour les conteneurs à clefs uniques et n'importe quelle valeur pour les conteneurs à clefs multiples. La complexité de cette méthode est `ln(N)+n`, où `N` est le nombre d'éléments stockés dans le conteneur et `n` est le nombre d'éléments dont la clef est égale à la valeur passée en paramètre. Le premier terme provient en effet de la recherche du premier élément disposant de cette propriété, et le deuxième des comparaisons qui suivent pour compter les éléments désignés par la clef.

Note : Les implémentations de la bibliothèque standard utilisent généralement la structure de données des arbres rouges et noirs pour implémenter les conteneurs associatifs. Cette structure algorithmique est une forme d'arbre binaire équilibré, dont la hauteur est au plus le logarithme binaire du nombre d'éléments contenus. Ceci explique les performances des conteneurs associatifs sur les opérations de recherche.

Chapitre 18. Les algorithmes

La plupart des opérations qui peuvent être appliquées aux structures de données ne sont pas spécifiques à ces structures. Par exemple, il est possible de trier quasiment toutes les séquences, que ce soient des listes, des vecteurs ou des deques. Les classes `template` des conteneurs de la bibliothèque standard ne fournissent donc que des méthodes de base permettant de les manipuler, et rares sont les conteneurs qui définissent des opérations dont le rôle dépasse le simple cadre de l'ajout, de la suppression ou de la recherche d'éléments. Au lieu de cela, la bibliothèque standard définit tout un jeu de fonctions `template` extérieures aux conteneurs et dont le but est de réaliser ces opérations de haut niveau. Ces fonctions sont appelées *algorithmes* en raison du fait qu'elles effectuent les traitements des algorithmes les plus connus et les plus utilisés en informatique.

Les algorithmes ne dérogent pas à la règle de généricité que la bibliothèque standard C++ s'impose. Autrement dit, ils sont capables de travailler en faisant le moins d'hypothèses possibles sur la structure de données contenant les objets sur lesquels ils s'appliquent. Ainsi, tous les algorithmes sont des fonctions `template` et ils travaillent sur les objets exclusivement par l'intermédiaire d'itérateurs et de foncteurs. Cela signifie que les algorithmes peuvent, en pratique, être utilisés sur n'importe quelle structure de données ou n'importe quel conteneur, pourvu que les préconditions imposées sur les itérateurs et le type des données manipulées soient respectées.

Comme pour les méthodes permettant de manipuler les conteneurs, les algorithmes sont décrits par leur sémantique et par leur complexité. Cela signifie que les implémentations de la bibliothèque standard sont libres quant à la manière de réaliser ces algorithmes, mais qu'elles doivent impérativement respecter les contraintes de performances imposées par la norme de la bibliothèque standard. En pratique cependant, tout comme pour les structures de données des conteneurs, ces contraintes imposent souvent l'algorithme sous-jacent pour l'implémentation de ces fonctionnalités et l'algorithme utilisé est le meilleur algorithme connu à ce jour. Autrement dit, les algorithmes de la bibliothèque standard sont forcément les plus efficaces qui soient.

La plupart des algorithmes de la bibliothèque standard sont déclarés dans l'en-tête `algorithm`. Certains algorithmes ont été toutefois définis initialement pour les valarray et n'apparaissent donc pas dans cet en-tête. Au lieu de cela, ils sont déclarés dans l'en-tête `numeric`. Ces algorithmes sont peu nombreux et cette particularité sera signalée dans leur description.

Le nombre des algorithmes définis par la bibliothèque standard est impressionnant et couvre sans doute tous les besoins courants des programmeurs. Il est donc difficile, en raison de cette grande diversité, de présenter les algorithmes de manière structurée. Cependant, les sections suivantes regroupent ces algorithmes en fonction de la nature des opérations qu'ils sont supposés effectuer. Ces opérations comprennent les opérations de manipulation générales des données, les recherches d'éléments selon des critères particuliers, les opérations de tri et de comparaison, et enfin les opérations de manipulation ensemblistes.

18.1. Opérations générales de manipulation des données

Les algorithmes généraux de manipulation des données permettent de réaliser toutes les opérations classiques de type création, copie, suppression et remplacement, mais également de modifier l'ordre des séquences d'éléments ainsi que d'appliquer un traitement sur chacun des éléments des conteneurs.

Certains algorithmes peuvent modifier soit les données contenues par les conteneurs sur lesquels ils travaillent, soit les conteneurs eux-mêmes. En général ces algorithmes travaillent sur place, c'est à dire qu'ils modifient les données du conteneur directement. Cependant, pour certains algorithmes, il

est possible de stocker les données modifiées dans un autre conteneur. Le conteneur source n'est donc pas modifié et les données, modifiées ou non, sont copiées dans le conteneur destination. En général, les versions des algorithmes capables de faire cette copie à la volée ne sont fournies que pour les algorithmes peu complexes car le coût de la copie peut dans ce cas être aussi grand ou plus grand que le coût du traitement des algorithmes eux-mêmes. Il est donc justifié pour ces algorithmes de donner la possibilité de réaliser la copie pendant leur traitement afin de permettre aux programmes d'optimiser les programmes selon les cas d'utilisation. Le nom des algorithmes qui réalisent une copie à la volée est le même nom que leur algorithme de base, mais suffixé par le mot « `_copy` ».

18.1.1. Opérations d'initialisation et de remplissage

Il existe deux méthodes permettant d'initialiser un conteneur ou de générer une série d'objets pour initialiser un conteneur. La première méthode ne permet que de générer plusieurs copies d'un même objet, que l'on spécifie par valeur, alors que la deuxième permet d'appeler une fonction de génération pour chaque objet à créer.

Les algorithmes de génération et d'initialisation sont déclarés de la manière suivante dans l'en-tête `algorithm` :

```
template <class ForwarIterator, class T>
void fill(ForwardIterator premier, ForwardIterator dernier, const T &valeur);

template <class OutputIterator, class T>
void fill_n(OutputIterator premier, Size nombre, const T &valeur);

tempalte <class ForwardIterator, class T, class Generator>
void generate(ForwardIterator premier, ForwardIterator dernier, Generator g);

template <class OutputIterator, class T, class Generator>
void generate_n(OutputIterator premier, Size nombre, Generator g);
```

Chaque algorithme est disponible sous deux formes différentes. La première utilise un couple d'itérateurs référençant le premier et le dernier élément à initialiser, et la deuxième n'utilise qu'un itérateur sur le premier élément et le nombre d'éléments à générer. Le dernier paramètre permet de préciser la valeur à affecter aux éléments du conteneur cible pour les algorithmes `fill` et `fill_n`, ou un foncteur permettant d'obtenir une nouvelle valeur à chaque invocation. Ce foncteur ne prend aucun paramètre et renvoie la nouvelle valeur de l'objet.

Exemple 18-1. Algorithme de génération d'objets et de remplissage d'un conteneur

```
#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>

using namespace std;

int compte()
{
    static int i = 0;
    return i++;
}
```

```
int main(void)
{
    // Crée une liste de 20 entiers consécutifs :
    typedef list<int> li;
    li l;
    generate_n(back_inserter(l), 20, compte);
    // Affiche la liste :
    li::iterator i = l.begin();
    while (i != l.end())
    {
        cout << *i << endl;
        ++i;
    }
    return 0;
}
```

Ces algorithmes effectuent exactement autant d'affectations qu'il y a d'éléments à créer ou à initialiser. Leur complexité est donc linéaire en fonction du nombre de ces éléments.

18.1.2. Opérations de copie

La bibliothèque standard définit deux algorithmes fondamentaux pour réaliser la copie des données des conteneurs. Ces algorithmes sont déclarés comme suit dans l'en-tête `algorithm` :

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator premier, InputIterator dernier,
    OutputIterator destination);

template <class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(
    BidirectionalIterator1 premier, BidirectionalIterator1 dernier,
    BidirectionalIterator2 fin_destination);
```

Alors que `copy` réalise la copie des objets référencés par les itérateurs `premier` et `dernier` du premier vers le dernier, l'algorithme `backward_copy` travaille dans le sens contraire. On utilisera donc typiquement `backward_copy` à chaque fois que la zone mémoire destination empiète sur la fin des données sources. Notez que dans ce cas, l'itérateur spécifiant la destination référence le dernier emplacement utilisé après la copie et non le premier élément. Autrement dit, l'itérateur `fin_destination` est utilisé de manière descendante, alors que l'itérateur `destination` fourni à l'algorithme `copy` est utilisé de manière ascendante.

Exemple 18-2. Algorithme de copie inverse

```
#include <iostream>
#include <algorithm>
#include <cstring>

using namespace std;

int main(void)
{
    char sBuffer[] = "abcdefg123";
    // Détermine l'itérateur de fin :
```

```
    char *pFin = sBuffer + strlen(sBuffer);
    // Écrase la chaîne par elle-même à partir du 'd' :
    copy_backward(sBuffer, pFin-3, pFin);
    // Affiche le résultat :
    cout << sBuffer << endl;
    return 0;
}
```

Note : La fonction `strlen` utilisée dans cet exemple est une des fonctions de la bibliothèque C standard, qui est déclarée dans l'en-tête `cstring`. Elle permet de calculer la longueur d'une chaîne de caractères C (sans compter le caratère nul terminal).

Ces algorithmes effectuent exactement autant d'affectation qu'il y a d'éléments à copier. Leur complexité est donc linéaire en fonction du nombre de ces éléments.

Note : Il existe également des algorithmes capables de réaliser une copie de leur résultat à la volée. Le nom de ces algorithmes est généralement le nom de leur algorithme de base suffixé par la chaîne `_copy`. Ces algorithmes seront décrits avec leurs algorithmes de base.

18.1.3. Opérations d'échange d'éléments

Il est possible d'échanger le contenu de deux séquences d'éléments grâce à un algorithme dédié à cette tâche, l'algorithme `swap_ranges`. Cet algorithme est déclaré comme suit dans l'en-tête `algorithm` :

```
template <class ForwardIterator, class ForwardIterator2>
ForwardIterator2 swap_ranges(ForwardIterator premier, ForwardIterator dernier,
    ForwardIterator2 destination);
```

Cet algorithme prend en paramètre les deux itérateurs définissant la première séquence et un itérateur destination permettant d'indiquer le premier élément de la deuxième séquence avec les éléments de laquelle l'échange doit être fait. La valeur retournée est l'itérateur de fin de cette séquence, une fois l'opération terminée.

Exemple 18-3. Algorithme d'échange

```
#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

int main(void)
{
    // Définit une liste d'entiers :
    typedef list<int> li;
    li l;
    l.push_back(2);
    l.push_back(5);
    l.push_back(3);
```

```
    l.push_back(7);
    // Définit un tableau de quatre éléments :
    int t[4] = {10, 11, 12, 13};
    // Échange le contenu du tableau et de la liste :
    swap_ranges(t, t+4, l.begin());
    // Affiche le tableau :
    int i;
    for (i=0; i<4; ++i)
        cout << t[i] << " ";
    cout << endl;
    // Affiche la liste :
    li::iterator it = l.begin();
    while (it != l.end())
    {
        cout << *it << " ";
        ++it;
    }
    cout << endl;
    return 0;
}
```

Cet algorithme n'échange pas plus d'éléments que nécessaire, autrement dit, il a une complexité linéaire en fonction de la taille de la séquence initiale.

18.1.4. Opérations de suppression d'éléments

Les conteneurs de la bibliothèque standard disposent tous de méthodes puissantes permettant d'effectuer des suppressions d'éléments selon différents critères. Toutefois, la bibliothèque standard définit également des algorithmes de suppression d'éléments dans des séquences. En fait, ces algorithmes n'effectuent pas à proprement parler de suppression, mais une réécriture des séquences au cours de laquelle les éléments à supprimer sont tout simplement ignorés. Ces algorithmes renvoient donc l'itérateur du dernier élément copié, au-delà duquel la séquence initiale est inchangée.

La bibliothèque standard fournit également des versions de ces algorithmes capables de réaliser une copie à la volée des éléments de la séquence résultat. Ces algorithmes peuvent donc typiquement être utilisés pour effectuer un filtre sur des éléments dont le but serait de supprimer les éléments indésirables.

Les fonctions de suppression des éléments sont déclarées comme suit dans l'en-tête `algorithm` :

```
template <class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator premier, ForwardIterator second,
    const T &valeur);

template <class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator premier, InputIterator second,
    OutputIterator destination, const T &valeur);

template <class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator premier, ForwardIterator second,
    Predicate p);

template <class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator premier, InputIterator second,
    OutputIterator destination, Predicate p);
```

Toutes ces fonctions prennent en premier et en deuxième paramètre les itérateurs définissant l'intervalle d'éléments sur lequel elles doivent travailler. Pour les fonctions de suppression d'un élément particulier, la valeur de cet élément doit également être fournie. Si vous préférez utiliser les versions basées sur un prédicat, il vous faut spécifier un foncteur unaire prenant en paramètre un élément et renvoyant un booléen indiquant si cet élément doit être supprimé ou non. Enfin, les versions de ces algorithmes permettant de réaliser une copie à la volée nécessitent bien entendu un itérateur supplémentaire indiquant l'emplacement destination où les éléments non supprimés devront être stockés.

Comme vous pouvez le constater d'après leurs déclarations, ces algorithmes renvoient tous un itérateur référençant l'élément suivant le dernier élément de la séquence résultat. Cet itérateur permet de déterminer la fin de la séquence d'éléments résultat, que cette séquence ait été modifiée sur place ou qu'une copie ait été réalisée. Si l'algorithme utilisé n'effectue pas de copie, les éléments suivant cet itérateur sont les éléments de la séquence initiale. C'est à ce niveau que la différence entre les algorithmes de suppression et les méthodes `erase` des conteneurs (et les méthodes `remove` des listes) apparaît : les algorithmes écrasent les éléments supprimés par les éléments qui les suivent, mais ne suppriment pas les éléments source du conteneur situés au-delà de l'itérateur renvoyé, alors que les méthodes `erase` des conteneurs suppriment effectivement des conteneurs les éléments à éliminer.

Exemple 18-4. Algorithme de suppression

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    // Construit un tableau de 10 entiers :
    int t[10] = { 1, 2, 2, 3, 5, 2, 4, 3, 6, 7 };
    // Supprime les entiers valant 2 :
    int *fin = remove(t, t+10, 2);
    // Affiche le tableau résultat :
    int *p = t;
    while (p != fin)
    {
        cout << *p << endl;
        ++p;
    }
    return 0;
}
```

De manière similaire, la bibliothèque standard définit également des algorithmes permettant de supprimer les doublons dans des séquences d'éléments. Ces algorithmes sont déclarés comme suit dans l'en-tête `algorithm` :

```
template<class ForwardIterator>
ForwardIterator unique(ForwardIterator premier, ForwardIterator dernier);

template<class ForwardIterator, class OutputIterator>
OutputIterator unique_copy(ForwardIterator premier, ForwardIterator dernier);

template <class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator premier, ForwardIterator dernier,
```

```
    BinaryPredicate p);

template <class ForwardIterator, class OutputIterator, class BinaryPredicate>
OutputIterator unique_copy(ForwardIterator premier, ForwardIterator dernier,
    BinaryPredicate p);
```

Ces algorithmes fonctionnent de la même manière que les algorithmes `remove` à ceci près qu'ils n'éliminent que les doublons dans la séquence source. Cela signifie qu'il n'est pas nécessaire de préciser la valeur des éléments à éliminer d'une part et, d'autre part, que les prédicats utilisés sont des prédicats binaires puisqu'ils doivent être appliqués aux couples d'éléments successifs.

> **Note :** Il n'existe pas d'algorithmes `unique_if` et `unique_copy_if`. La bibliothèque standard utilise les possibilités de surcharge du C++ pour distinguer les versions avec et sans prédicat des algorithmes de suppression des doublons.

Exemple 18-5. Algorithme de suppression des doublons

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    // Construit un tableau de 10 entiers :
    int t[10] = { 1, 2, 2, 3, 5, 2, 4, 3, 6, 7 };
    // Supprime les doublons :
    int *fin = unique(t, t+10);
    // Affiche le tableau résultat :
    int *p = t;
    while (p != fin)
    {
        cout << *p << endl;
        ++p;
    }
    return 0;
}
```

Le test de suppression est appliqué par ces algorithmes autant de fois qu'il y a d'éléments dans la séquence initiale, c'est-à-dire que leur complexité est linéaire en fonction du nombre d'éléments de cette séquence.

18.1.5. Opérations de remplacement

Les algorithmes de remplacement permettent de remplacer tous les éléments d'un conteneur vérifiant une propriété particulière par un autre élément dont la valeur doit être fournie en paramètre. Les éléments devant être remplacés peuvent être identifiés soit par leur valeur, soit par un prédicat unaire prenant en paramètre un élément et renvoyant un booléen indiquant si cet élément doit être remplacé ou non. Les algorithmes de remplacement sont déclarés comme suit dans l'en-tête `algorithm` :

```
template <class ForwardIterator, class T>
```

```
void replace(ForwardIterator premier, ForwardIterator dernier,
    const T &ancienne_valeur, const T &nouvelle_valeur);

template <class InputIterator, class OutputIterator, class T>
void replace_copy(InputIterator premier, InputIterator dernier,
    OutputIterator destination,
    const T &ancienne_valeur, const T &nouvelle_valeur);

template <class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator premier, ForwardIterator dernier,
    Predicate p, const T &nouvelle_valeur);

template <class InputIterator, class OutputIterator,
    class Predicate, class T>
void replace_copy_if(InputIterator premier, InputIterator dernier,
    OutputIterator destination,
    Predicate p, const T &nouvelle_valeur);
```

Les algorithmes de remplacement peuvent travailler sur place ou effectuer une copie à la volée des éléments sur lesquels ils travaillent. Les versions capables de réaliser ces copies sont identifiées par le suffixe `_copy` de leur nom. Ces algorithmes prennent un paramètre supplémentaire permettant de spécifier l'emplacement destination où les éléments copiés devront être stockés. Ce paramètre est un itérateur, tout comme les paramètres qui indiquent l'intervalle d'éléments dans lequel la recherche et le remplacement doivent être réalisés.

Exemple 18-6. Algorithme de recherche et de remplacement

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    int t[10] = {1, 2, 5, 3, 2, 7, 6, 4, 2, 1};
    // Remplace tous les 2 par des 9 :
    replace(t, t+10, 2, 9);
    // Affiche le résultat :
    int i;
    for (i=0; i<10; ++i)
        cout << t[i] << endl;
    return 0;
}
```

Le test de remplacement est appliqué par ces algorithmes autant de fois qu'il y a des éléments dans la séquence initiale, c'est-à-dire que leur complexité est linéaire en fonction du nombre d'éléments de cette séquence.

18.1.6. Réorganisation de séquences

Comme il l'a été expliqué dans la Section 17.2, l'ordre des éléments d'une séquence est important. La plupart des séquences conservent les éléments dans l'ordre dans lequel ils ont été insérés, d'autre se réorganisent automatiquement lorsque l'on travaille dessus pour assurer un ordre bien défini.

La bibliothèque standard fournit plusieurs algorithmes permettant de réorganiser la séquence des éléments dans un conteneur qui ne prend pas en charge lui-même l'ordre de ses éléments. Ces algorithmes permettent de réaliser des rotations et des permutations des éléments, des symétries et des inversions, ainsi que de les mélanger de manière aléatoire.

> **Note :** Il existe également des algorithmes de tri extrêmement efficaces, mais ces algorithmes seront décrits plus loin dans une section qui leur est consacrée.

18.1.6.1. Opérations de rotation et de permutation

Les algorithmes de rotation permettent de faire tourner les différents éléments d'une séquence dans un sens ou dans l'autre. Par exemple, dans une rotation vers la gauche d'une place, le deuxième élément peut prendre la place du premier, le troisième celle du deuxième, etc., le premier élément revenant à la place du dernier. Ces algorithmes sont déclarés de la manière suivante dans l'en-tête `algorithm` :

```
template <class ForwardIterator>
void rotate(ForwardIterator premier, ForwardIterator pivot,
    ForwardIterator dernier);

template <class ForwardIterator, class OutputIterator>
void rotate_copy(ForwardIterator premier, ForwardIterator pivot,
    ForwardIterator dernier, OutputIterator destination);
```

Les algorithmes de rotation prennent en paramètre un itérateur indiquant le premier élément de la séquence devant subir la rotation, un itérateur référençant l'élément qui se trouvera en première position après la rotation, et un itérateur référençant l'élément suivant le dernier élément de la séquence. Ainsi, pour effectuer une rotation d'une position vers la gauche, il suffit d'utiliser pour l'itérateur `pivot` la valeur de l'itérateur suivant l'itérateur `premier` et, pour effectuer une rotation d'une position vers la droite, il faut prendre pour l'itérateur `pivot` la valeur précédant celle de l'itérateur `dernier`.

Exemple 18-7. Algorithme de rotation

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    int t[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    // Effectue une rotation pour amener le quatrième
    // élément en première position :
    rotate(t, t+3, t+10);
    // Affiche le résultat :
    int i;
    for (i=0; i<10; ++i)
        cout << t[i] << endl;
    return 0;
}
```

La bibliothèque fournit également des algorithmes permettant d'obtenir l'ensemble des permutations d'une séquence d'éléments. Rappelons qu'une permutation est une des combinaisons possibles des

valeurs des différents éléments d'un ensemble, en considérant les éléments d'égale valeur comme identiques. Par exemple, si un ensemble contient deux éléments de même valeur, il n'y a qu'une seule permutation possible : les deux valeurs. Si en revanche ces deux éléments ont deux valeurs distinctes, on peut réaliser deux permutations selon la valeur que l'on place en premier.

Les algorithmes de permutation de la bibliothèque ne permettent pas d'obtenir les permutations directement. Au lieu de cela, ils permettent de passer d'une permutation à la permutation suivante ou à la précédente. Cela suppose qu'une relation d'ordre soit définie sur l'ensemble des permutations de la séquence. La bibliothèque standard utilise l'ordre lexicographique pour classer ces permutations. Autrement dit, les premières permutations sont celles pour lesquelles les premiers éléments ont les valeurs les plus faibles.

Les algorithmes de calcul des permutations suivante et précédente sont déclarés comme suit dans l'en-tête `algorithm` :

```
template <class BidirectionalIterator>
bool next_permutation(BidirectionalIterator premier, BidirectionalIterator dernier);

template <class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator premier, BidirectionalIterator dernier);
```

Ces algorithmes prennent tous les deux deux itérateurs indiquant les éléments devant subir la permutation et renvoient un booléen indiquant si la permutation suivante ou précédente existe ou non. Si ces permutations n'existent pas, les algorithmes `next_permutation` et `prev_permutation` bouclent et calculent respectivement la plus petite et la plus grande permutation de l'ensemble des permutations.

Exemple 18-8. Algorithme de permutation

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    int t[3] = {1, 1, 2};
    // Affiche l'ensemble des permutations de (1, 1, 2) :
    do
    {
        int i;
        for (i=0; i<3; ++i)
            cout << t[i] << " ";
        cout << endl;
    }
    while (next_permutation(t, t+3));
    return 0;
}
```

Les algorithmes de rotation effectuent autant d'échange qu'il y a d'éléments dans la séquence initiale, et les algorithmes de calcul de permutation en font exactement la moitié. La complexité de ces algorithmes est donc linéaire en fonction du nombre d'éléments de l'intervalle qui doit subir l'opération.

18.1.6.2. Opérations d'inversion

Il est possible d'inverser l'ordre des éléments d'une séquence à l'aide des algorithmes `reverse` et `reverse_copy`. Ces algorithmes sont déclarés de la manière suivante dans l'en-tête `algorithm` :

```
template <class BidirectionalIterator>
void reverse(BidirectionalIterator premier, BidirectionalIterator dernier);

template <class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator premier,
    BidirectionalIterator dernier, OutputIterator destination);
```

Ces algorithmes prennent en paramètre les itérateurs permettant de spécifier l'intervalle des éléments qui doit être inversé. La version de cet algorithme qui permet de réaliser une copie prend un paramètre supplémentaire qui doit recevoir l'itérateur référençant l'emplacement destination dans lequel le résultat de l'inversion doit être stocké. Cet itérateur retourne la valeur de l'itérateur destination passé le dernier élément écrit.

Exemple 18-9. Algorithme d'inversion

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    int t[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    // Inverse le tableau :
    reverse(t, t+10);
    // Affiche le résultat :
    int i;
    for (i=0; i<10; ++i)
        cout << t[i] << endl;
    return 0;
}
```

Les algorithmes d'inversion effectuent autant d'échange d'éléments qu'il y en a dans la séquence initiale. Autrement dit, leur complexité est linéaire en fonction de la taille de cette séquence.

18.1.6.3. Opérations de mélange

Il est possible de redistribuer aléatoirement les éléments d'une séquence à l'aide de l'algorithme `random_shuffle`. Cet algorithme est fourni sous la forme de deux surcharges déclarées comme suit dans l'en-tête `algorithm` :

```
template <class RandomAccessIterator>
void random_shuffle(RandomAccessIterator premier, RandomAccessIterator dernier);

template <class RandomAccessIterator, class RandomNumberGenerator>
void random_shuffle(RandomAccessIterator premier, RandomAccessIterator dernier,
    RandomNumberGenerator g);
```

Ces algorithmes prennent en paramètre les itérateurs de début et de fin de la séquence dont les éléments doivent être mélangés. La deuxième version de cet algorithme peut prendre en dernier paramètre un foncteur qui sera utilisé pour calculer les positions des éléments pendant le mélange. Ainsi, cette surcharge permet de spécifier soi-même la fonction de distribution à utiliser pour effectuer cette nouvelle répartition. Ce foncteur doit prendre en paramètre une valeur du type difference_type des itérateurs utilisés pour référencer les éléments de la séquence, et renvoyer une valeur comprise entre 0 et la valeur reçue en paramètre. Il doit donc se comporter comme la fonction `rand` de la bibliothèque standard C (déclarée dans le fichier d'en-tête `cstdlib`).

Exemple 18-10. Algorithme de mélange

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    int t[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    // Mélange le tableau t :
    random_shuffle(t, t+10);
    // Affiche le résultat :
    int i;
    for (i=0; i<10; ++i)
        cout << t[i] << endl;
    return 0;
}
```

Ces algorithmes effectuent exactement le nombre d'éléments de la séquence à mélanger moins un échanges de valeurs. Leur complexité est donc linéaire en fonction du nombre d'éléments de ces séquences.

18.1.7. Algorithmes d'itération et de transformation

Les algorithmes de transformation et d'itération de la bibliothèque standard font partie des plus utiles puisqu'ils permettent d'effectuer un traitement sur l'ensemble des éléments d'un conteneur. Ces traitements peuvent modifier ou non ces éléments ou tout simplement calculer une valeur à partir de ces éléments.

Les deux principaux algorithmes fournis par la bibliothèque standard sont sans doute les algorithmes `for_each` et `transform`, qui permettent d'effectuer une action sur chaque élément d'un conteneur. Ces algorithmes sont déclarés comme suit dans l'en-tête `algorithm` :

```
template <class InputIterator, class Function>
Function for_each(InputIterator premier, InputIterator dernier, Function f);

template <class InputIterator, class OutputIterator,
    class UnaryOperation>
OutputIterator transform(InputIterator premier, InputIterator dernier,
    OutputIterator destination, UnaryOperation op);

template <class InputIterator1, class InputIterator2,
    class OutputIterator, class BinaryOperation>
OutputIterator transform(InputIterator1 premier1, InputIterator1 dernier1,
```

```
    InputIterator2 premier2, OutputIterator destination,
    BinaryOperation op);
```

L'algorithme `for_each` permet d'itérer les éléments d'un conteneur et d'appeler une fonction pour chacun de ces éléments. Il prend donc en paramètre deux itérateurs permettant de spécifier les éléments à itérer et un foncteur qui sera appelé à chaque itération. Pendant l'itération, ce foncteur reçoit en paramètre la valeur de l'élément de l'itération courante de la part de `for_each`. Cette valeur ne doit en aucun cas être modifiée et la valeur retournée par ce foncteur est ignorée. La valeur retournée par l'algorithme `for_each` est le foncteur qui lui a été communiqué en paramètre.

Contrairement à `for_each`, qui ne permet pas de modifier les éléments qu'il itère, l'algorithme `transform` autorise la modification des éléments du conteneur sur lequel il travaille. Il est fourni sous deux versions, la première permettant d'appliquer un foncteur unaire sur chaque élément d'un conteneur et la deuxième un foncteur binaire sur deux éléments de deux conteneurs sur lesquels l'algorithme itère simultanément. Les deux versions prennent en premiers paramètres les itérateurs permettant de spécifier les éléments à itérer du premier conteneur. Ils prennent également en paramètre un foncteur permettant de calculer une nouvelle valeur à partir des éléments itérés et un itérateur dans lequel les résultats de ce foncteur seront stockés. La version permettant de travailler avec un foncteur binaire prend un paramètre complémentaire, qui doit recevoir la valeur de l'itérateur de début du conteneur devant fournir les éléments utilisés en tant que second opérande du foncteur. La valeur retournée par les algorithmes `transform` est la valeur de fin de l'itérateur destination.

Exemple 18-11. Algorithmes d'itération

```
#include <iostream>
#include <functional>
#include <algorithm>

using namespace std;

void aff_entier(int i)
{
    cout << i << endl;
}

int main(void)
{
    int t[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    // Inverse tous les éléments du tableau :
    transform(t, t+10, t, negate<int>());
    // Affiche le résultat :
    for_each(t, t+10, ptr_fun(&aff_entier));
    return 0;
}
```

Comme vous pouvez le constater d'après cet exemple, il est tout à fait possible d'utiliser la même valeur pour l'itérateur `premier` et l'itérateur `destination`. Cela signifie que les éléments itérés peuvent être remplacés par les nouvelles valeurs calculées par le foncteur fourni à l'algorithme `transform`.

Un cas particulier des algorithmes d'itération est celui des algorithmes `count` et `count_if` puisque le traitement effectué est alors simplement le décompte des éléments vérifiant une certaine condition. Ces deux algorithmes permettent en effet de compter le nombre d'éléments d'un conteneur dont la

valeur est égale à une valeur donnée ou vérifiant un critère spécifié par l'intermédiaire d'un prédicat unaire. Ces deux algorithmes sont déclarés de la manière suivante dans l'en-tête `algorithm` :

```
template <class InputIterator, class T>
iterator_traits<InputIterator>::difference_type
    count(InputIterator premier, InputIterator dernier, const T &valeur);

template <class InputIterator, class Predicate>
iterator_traits<InputIterator>::difference_type
    count_if(InputIterator premier, InputIterator dernier, Predicate p);
```

Comme vous pouvez le constater, ces algorithmes prennent en paramètre deux itérateurs spécifiant l'intervalle des éléments sur lesquels le test doit être effectué, et la valeur avec laquelle ces éléments doivent être comparés ou un prédicat unaire. Dans ce cas, le résultat de ce prédicat indique si l'élément qu'il reçoit en paramètre doit être compté ou non.

Exemple 18-12. Algorithme de décompte d'éléments

```
#include <iostream>
#include <functional>
#include <algorithm>

using namespace std;

bool parity_even(int i)
{
    return (i & 1) == 0;
}

int main(void)
{
    int t[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    // Compte le nombre d'éléments pairs :
    cout << count_if(t, t+10, ptr_fun(&parity_even)) << endl;
    return 0;
}
```

Tous les algorithmes d'itération ne font qu'un seul passage sur chaque élément itéré. Autrement dit, la complexité de ces algorithmes est linéaire en fonction du nombre d'éléments compris entre les deux itérateurs spécifiant l'intervalle d'éléments sur lequel ils sont appliqués.

Enfin, la bibliothèque standard fournit des algorithmes de calcul plus évolués, capables de travailler sur les éléments des conteneurs. Ces algorithmes sont généralement utilisés en calcul numérique et ont été conçus spécialement pour les tableaux de valeurs. Cependant, ils restent tout à fait utilisables sur d'autres conteneurs que les valarray, la seule distinction qu'ils ont avec les autres algorithmes de la bibliothèque standard est qu'ils sont déclarés dans l'en-tête `numeric` au lieu de l'en-tête `algorithm`. Ces algorithmes sont les suivants :

```
template <class InputIterator, class T>
T accumulate(InputIterator premier, InputIterator dernier, T init);

template <class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator premier, InputIterator dernier,
    T init, BinaryOperation op);
```

```
template <class InputIterator1, class InputIterator2, class T>
T inner_product(InputIterator1 premier1, InputIterator1 dernier1,
    InputIterator2 premier2, T init);

template <class InputIterator1, class InputIterator2, class T,
    class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIterator1 premier1, InputIterator1 dernier1,
    InputIterator2 premier2, T init,
    BinaryOperation1 op1, BinaryOperation op2);

template <class InputIterator, class OutputIterator>
OutputIterator partial_sum(InputIterator premier, InputIterator dernier,
    OutputIterator destination);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator partial_sum(InputIterator premier, InputIterator dernier,
    OutputIterator destination, BinaryOperation op);

template <class InputIterator, class OutputIterator>
OutputIterator adjacent_difference(InputIterator premier, InputIterator dernier,
    OutputIterator destination);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator adjacent_difference(InputIterator premier, InputIterator dernier,
    OutputIterator destination, BinaryOperation op);
```

Ces algorithmes correspondent à des opérations courantes, que l'on fait généralement sur les tableaux de nombres de type valarray. L'algorithme `accumulate` permet généralement de réaliser la somme des valeurs qui sont stockées dans un conteneur. L'algorithme `inner_product` est utilisé quant à lui pour réaliser le *produit scalaire* de deux séquences de nombres, opération mathématique généralement effectuée dans le calcul vectoriel. Enfin, les algorithmes `partial_sum` et `adjacent_difference` réalisent respectivement le calcul des sommes partielles et des différences deux à deux des éléments d'un conteneur.

Pout tous ces algorithmes, il est possible d'utiliser d'autres opérations que les opérations généralement utilisées. Par exemple, `accumulate` peut utiliser une autre opération que l'addition pour « accumuler » les valeurs des éléments. Pour cela, la bibliothèque standard fournit des surcharges de ces algorithmes capables de travailler avec des foncteurs binaires. Ces foncteurs doivent accepter deux paramètres du type des éléments du conteneur sur lequel les algorithmes sont appliqués et renvoyer une valeur du même type, calculée à partir de ces paramètres.

L'algorithme `accumulate` prend donc en premiers paramètres les itérateurs définissant l'intervalle des valeurs qui doivent être accumulées. Il initialise la valeur d'une variable *accumulateur* avec la valeur fournie en troisième paramètre, et parcours l'ensemble des éléments. Pour chaque élément traité, `accumulate` remplace la valeur courante de l'accumulateur par le résultat de l'opération d'accumulation appliquée à l'accumulateur lui-même et à la valeur de l'élément courant. Par défaut, l'opération d'accumulation utilisée est l'addition, mais il est possible de changer ce comportement en fournissant un foncteur binaire en dernier paramètre. Lorsque l'ensemble des éléments a été parcouru, la valeur de l'accumulateur est retournée.

Exemple 18-13. Algorithme d'accumulation

```
#include <list>
#include <numeric>
```

```
#include <functional>
#include <iostream>

using namespace std;

int main(void)
{
    // Construit une liste d'entiers :
    typedef list<int> li;
    li l;
    l.push_back(5);
    l.push_back(2);
    l.push_back(9);
    l.push_back(1);
    // Calcule le produit de ces entiers :
    int res = accumulate(l.begin(), l.end(),
        1, multiplies<int>());
    cout << res << endl;
    return 0;
}
```

L'algorithme `inner_product` travaille sur deux conteneurs simultanément et réalise leur produit scalaire. Le produit scalaire est l'opération qui consiste à multiplier les éléments de deux séries de nombres deux à deux, et de faire la somme des résultats. L'algorithme `inner_product` prend donc en paramètre les itérateurs de début et de fin spécifiant la première série de nombres, l'itérateur de début de la deuxième série de nombres, et la valeur initiale de l'accumulateur utilisé pour réaliser la somme des produits des éléments de ces deux conteneurs. Bien entendu, tout comme pour l'algorithme `accumulate`, il est possible de remplacer les opérations de multiplication et d'addition de l'algorithme standard par deux foncteurs en fournissant ceux-ci en derniers paramètres.

Exemple 18-14. Algorithme de produit scalaire

```
#include <iostream>
#include <numeric>

using namespace std;

int main(void)
{
    // Définit deux vecteurs orthogonaux :
    int t1[3] = {0, 1, 0};
    int t2[3] = {0, 0, 1};
    // Calcule leur produit scalaire :
    int res = inner_product(t1, t1+3, t2, 0);
    // Le produit scalaire de deux vecteurs orthogonaux
    // est toujours nul :
    cout << res << endl;
    return 0;
}
```

L'algorithme `partial_sum` permet de calculer la série des sommes partielles de la suite de valeurs spécifiée par les deux itérateurs fournis en premiers paramètres. Cette série de sommes contiendra d'abord la valeur du premier élément, puis la valeur de la somme des deux premiers éléments, puis la valeur de la somme des trois premiers éléments, etc., et enfin la somme de l'ensemble des éléments de la suite de valeurs sur laquelle l'algorithme travaille. Toutes ces valeurs sont stockées successivement aux emplacements indiqués par l'itérateur `destination`. Comme pour les autres algorithmes, il est

possible de spécifier une autre opération que l'addition à l'aide d'un foncteur binaire que l'on passera en dernier paramètre.

Enfin, l'algorithme `adjacent_difference` est l'algorithme inverse de l'algorithme `parial_sum`. En effet, il permet de calculer la série des différences des valeurs des éléments successifs d'une suite de valeurs, pris deux à deux. Cet algorithme prend en paramètre les itérateurs décrivant la suite de valeurs sur laquelle il doit travailler, l'itérateur de l'emplacement destination où les résultats devront être stockés et éventuellement le foncteur à appliquer aux couples d'éléments successifs traités par l'algorithme. La première différence est calculée en supposant que l'élément précédent le premier élément a pour valeur la valeur nulle. Ainsi, le premier élément de l'emplacement destination est toujours égal au premier élément de la suite de valeurs sur laquelle l'algorithme travaille.

Exemple 18-15. Algorithmes de sommes partielles et de différences adjacentes

```
#include <iostream>
#include <numeric>

using namespace std;

int main(void)
{
    int t[4] = {1, 1, 1, 1};
    // Calcule les sommes partielles des éléments
    // du tableau :
    partial_sum(t, t+4, t);
    // Affiche le résultat :
    int i;
    for (i=0; i<4; ++i)
        cout << t[i] << " ";
    cout << endl;
    // Calcule les différences adjacentes :
    adjacent_difference(t, t+4, t);
    // C'est le tableau initial :
    for (i=0; i<4; ++i)
        cout << t[i] << " ";
    cout << endl;
    return 0;
}
```

Tous ces algorithmes travaillent en une seule passe sur les éléments des conteneurs sur lesquels ils s'appliquent. Leur complexité est donc linéaire en fonction du nombre d'éléments spécifiés par les itérateurs fournis en premier paramètre.

18.2. Opérations de recherche

En général, la plupart des opérations de recherche de motifs que les programmes sont susceptibles d'effectuer se font sur des chaînes de caractères ou sur les conteneurs associatifs. Cependant, il peut être nécessaire de rechercher un élément dans un conteneur selon un critère particulier ou de rechercher une séquence d'éléments constituant un motif à retrouver dans la suite des éléments d'un conteneur. Enfin, il est relativement courant d'avoir à rechercher les groupes d'éléments consécutifs disposant de la même valeur dans un conteneur. Toutes ces opérations peuvent être réalisées à l'aide des algorithmes de recherche que la bibliothèque standard met à la disposition des programmeurs.

18.2.1. Opération de recherche d'éléments

Le premier groupe d'opérations de recherche contient tous les algorithmes permettant de retrouver un élément dans un conteneur, en l'identifiant soit par sa valeur, soit par une propriété particulière. Toutefois, cet élément peut ne pas être le seul élément vérifiant ce critère. La bibliothèque standard définit donc plusieurs algorithmes permettant de rechercher ces éléments de différentes manières, facilitant ainsi les opérations de recherche dans différents contextes.

Les algorithmes de recherche d'éléments sont les algorithmes `find` et `find_if`, qui permettent de retrouver le premier élément d'un conteneur vérifiant une propriété particulière, et l'algorithme `find_first_of`, qui permet de retrouver le premier élément vérifiant une relation avec une valeur parmi un ensemble de valeurs données. Tous ces algorithmes sont déclarés dans l'en-tête `algorithm` :

```
template <class InputIterator, class T>
InputIterator find(InputIterator premier, InputIterator dernier, const T &valeur);

template <class InputIterator, class Predicate>
InputIterator find_if(InputIterator premier, InputIterator dernier, Predicate p);

template <class InputIterator, class ForwardIterator>
InputIterator find_first_of(InputIterator premier1, InputIterator dernier1,
    ForwardIterator premier2, ForwardIterator dernier2);

template <class InputIterator, class ForwardIterator, class BinaryPredicate>
InputIterator find_first_of(InputIterator premier1, InputIterator dernier1,
    ForwardIterator premier2, ForwardIterator dernier2,
    BinaryPredicate p);
```

L'algorithme `find` prend en paramètre les deux itérateurs classiques définissant la séquence d'éléments dans laquelle la recherche doit être effectuée. Il prend également en paramètre la valeur de l'élément recherché, et renvoie un itérateur sur le premier élément qui dispose de cette valeur. Si vous désirez effectuer une recherche sur un autre critère que l'égalité des valeurs, vous devez utiliser l'algorithme `find_if`. Celui-ci prend un prédicat en paramètre à la place de la valeur. C'est la valeur de ce prédicat, appliqué à l'élément courant dans le parcours des éléments de la séquence définie par les itérateurs `premier` et `dernier`, qui permettra de déterminer si cet élément est celui recherché ou non. La valeur retournée est l'itérateur `dernier` si aucun élément ne correspond au critère de recherche. La complexité de cet algorithme est linéaire en fonction du nombre d'éléments de la séquence d'éléments dans laquelle la recherche se fait.

L'algorithme `find_first_of` prend deux couples d'itérateurs en paramètre. Le premier définit l'intervalle d'éléments dans lequel la recherche doit être effectuée et le deuxième un ensemble de valeur dont les éléments doivent être recherchés. L'algorithme renvoie un itérateur sur le premier élément qui est égal à l'une des valeurs de l'ensemble de valeurs spécifié par le deuxième couple d'itérateurs, ou l'itérateur `dernier1` si cet élément n'existe pas. Il est également possible d'utiliser un autre critère que l'égalité avec l'un des éléments de cet ensemble en utilisant un prédicat binaire dont la valeur indiquera si l'élément courant vérifie le critère de recherche. Lorsqu'il est appelé par l'algorithme, ce prédicat reçoit en paramètre l'élément courant de la recherche et l'une des valeurs de l'ensemble de valeurs spécifié par les itérateurs `premier2` et `dernier2`. La complexité de cet algorithme est $n \times m$, où n est le nombre d'éléments de la séquence dans laquelle la recherche est effectuée et m est le nombre de valeurs avec lesquelles ces éléments doivent être comparés.

Exemple 18-16. Algorithme de recherche d'éléments

```
#include <iostream>
```

```
#include <algorithm>

using namespace std;

int main(void)
{
    int t[10] = {0, 5, 3, 4, 255, 7, 0, 5, 255, 9};
    // Recherche les éléments valant 0 ou 255 :
    int sep[2] = {0, 255};
    int *debut = t;
    int *fin = t+10;
    int *courant;
    while ((courant=find_first_of(debut, fin,
        sep, sep+2)) != fin)
    {
        // Affiche la position de l'élément trouvé :
        cout << *courant << " en position " <<
            courant-t << endl;
        debut = courant+1;
    }
    return 0;
}
```

18.2.2. Opérations de recherche de motifs

Les opérations de recherche de motifs permettent de trouver les premières et les dernières occurrences d'un motif donné dans une suite de valeurs. Ces opérations sont réalisées respectivement par les algorithmes `search` et `find_end`, dont la déclaration dans l'en-tête `algorithm` est la suivante :

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(ForwardIterator1 premier1, ForwardIterator1 dernier1,
    ForwardIterator2 premier2, ForwardIterator2 dernier2);

template <class ForwardIterator1, class ForwardIterator2,
    class BinaryPredicate>
ForwardIterator1 search(ForwardIterator1 premier1, ForwardIterator1 dernier1,
    ForwardIterator2 premier2, ForwardIterator2 dernier2,
    BinaryPredicate p);

template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end(ForwardIterator1 premier1, ForwardIterator1 dernier1,
    ForwardIterator2 premier2, ForwardIterator2 dernier2);

template <class ForwardIterator1, class ForwardIterator2,
    class BinaryPredicate>
ForwardIterator1 find_end(ForwardIterator1 premier1, ForwardIterator1 dernier1,
    ForwardIterator2 premier2, ForwardIterator2 dernier2,
    BinaryPredicate p);
```

Tous ces algorithmes prennent en paramètre deux couples d'itérateurs, le premier permettant d'identifier la séquence des valeurs dans laquelle la recherche du motif doit être effectuée et le deuxième permettant d'identifier le motif lui-même. Chaque algorithme est fourni sous la forme de deux surcharges. La première recherche le motif en comparant les éléments à l'aide de l'opérateur

d'égalité du type des éléments comparés. La deuxième permet d'effectuer cette comparaison à l'aide d'un prédicat binaire, que l'on fournit dans ce cas en dernier paramètre.

La valeur retournée par l'algorithme `search` est un itérateur sur la première occurrence du motif dans la séquence de valeurs spécifiées par les itérateurs `premier1` et `dernier1` ou l'itérateur `dernier1` lui-même si ce motif n'y apparaît pas. De même, la valeur retournée par l'algorithme `find_end` est un itérateur référençant la dernière occurrence du motif dans la séquence des valeurs spécifiée par les itérateurs `premier1` et `dernier1`, ou l'itérateur `dernier1` lui-même si le motif n'a pas pu être trouvé.

Exemple 18-17. Algorithmes de recherche de motif

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    int t[10] = {1, 2, 4, 5, 3, 1, 2, 3, 5, 9};
    // Recherche le motif {1, 2, 3} dans le tableau :
    int motif[3] = {1, 2, 3};
    int *p = search(t, t+10, motif, motif+3);
    cout << "{1, 2, 3} en position " <<
        p - t << endl;
    // Recherche la dernière occurrence de {1, 2} :
    p = find_end(t, t+10, motif, motif+2);
    cout << "Dernier {1, 2} en position " <<
        p - t << endl;
    return 0;
}
```

La complexité de l'algorithme `search` est `n×m`, où `n` est le nombre d'éléments de la séquence spécifiée par le premier couple d'itérateurs et `m` est la longueur du motif à rechercher. La complexité de l'algorithme `find_end` est `n×(n-m)`.

Lorsque tous les éléments du motif sont égaux, il est possible d'utiliser l'algorithme `search_n`. Cet algorithme permet en effet de rechercher une série de valeurs identiques dans une séquence. Il est déclaré comme suit dans l'en-tête `algorithm` :

```
template <class ForwardIterator, class Size, class T>
ForwardIterator search_n(ForwardIterator premier, ForwardIterator dernier,
    Size nombre, const T &valeur);

template <class ForwardIterator, class Size, class T,
    class BinaryPredicate>
ForwardIterator search_n(ForwardIterator premier, ForwardIterator dernier,
    Size nombre, const T &valeur, BinaryPredicate p);
```

Les deux surcharges de cet algorithme prennent en paramètre les itérateurs définissant la séquence de valeurs dans laquelle la recherche doit être effectuée, la longueur du motif à rechercher, et la valeur des éléments de ce motif. La deuxième version de cet algorithme accepte également un prédicat binaire, qui sera utilisé pour effectuer la comparaison des éléments de la séquence dans laquelle la recherche se fait avec la valeur passée en paramètre. La valeur retournée est un itérateur référençant la première occurrence du motif recherché ou l'itérateur `dernier` si ce motif n'existe pas dans la séquence de

valeurs analysée. La complexité de l'algorithme `search_n` est `n×m`, où `n` est la taille de la séquence dans laquelle la recherche est effectuée et `m` est la longueur du motif recherché.

Un cas particulier de la recherche de valeurs successives est l'identification de doublons de valeurs. Cette identification peut être réalisée grâce à l'algorithme `adjacent_find`. Contrairement à l'algorithme `search_n`, `adjacent_find` localise tous les couples de valeurs d'une série de valeurs, quelle que soit la valeur des éléments de ces couples. Il est donc inutile de préciser cette valeur, et les surcharges de cet algorithme sont déclarées comme suit dans l'en-tête `algorithm` :

```
template <class ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator premier, ForwardIterator dernier);

template <class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator premier, ForwardIterator dernier,
      BinaryPredicate p);
```

Les itérateurs fournis en paramètre permettent comme à l'accoutumée de définir la séquence d'éléments dans laquelle la recherche s'effectue. La deuxième surcharge prend également en paramètre un prédicat binaire définissant la relation de comparaison utilisée par l'algorithme. La valeur retournée par ces algorithmes est l'itérateur référençant le premier doublon dans la séquence de valeurs analysée, ou l'itérateur `dernier` si aucun doublon n'existe.

Exemple 18-18. Algorithme de recherche de doublons

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    int t[10] = {0, 1, 2, 2, 3, 4, 4, 5, 6, 2};
    // Recherche les doublons dans le tableau :
    int *debut = t;
    int *fin = t+10;
    int *p;
    while ((p = adjacent_find(debut, fin)) != fin)
    {
        cout << "Doublon en position " << p-t << endl;
        debut = p+1;
    }
    return 0;
}
```

La complexité de cet algorithme est linéaire en fonction de la taille de la séquence de valeurs dans laquelle la recherche se fait.

18.3. Opérations d'ordonnancement

La bibliothèque standard fournit plusieurs algorithmes relatifs à l'ordonnancement des éléments dans les conteneurs. Grâce à ces algorithmes, il est possible de réorganiser la séquence de ces éléments

de manière à obtenir certaines propriétés basées sur la relation d'ordre. Ces réorganisations ont généralement pour but soit de trier complètement ces séquences, soit d'effectuer des tris partiels à partir desquels il est possible d'obtenir des informations relatives à l'ordre des éléments de manière très efficace.

La plupart des algorithmes de tri et d'ordonnancement se basent sur une structure de données très performante : les « `tas` ». Les algorithmes de manipulation de ces structures de données seront donc décrits en premier. Les sections qui suivront traiteront ensuite des algorithmes de tri et de recherches binaires dans un ensemble d'éléments déjà trié.

18.3.1. Opérations de gestion des tas

Un *tas* (« heap » en anglais) est une structure de données récursive dans laquelle le premier élément est toujours le plus grand élément et qui dispose d'une opération de suppression du premier élément ainsi que d'une opération d'ajout d'un nouvel élément extrêmement performantes. Plus précisément, les propriétés fondamentales des tas sont les suivantes :

- le premier élément du tas est toujours le plus grand de tous les éléments contenus ;
- il est possible de supprimer ce premier élément et cette opération de suppression a une complexité logarithmique en fonction du nombre d'éléments dans le tas ;
- il est possible d'ajouter un nouvel élément dans le tas avec une complexité également logarithmique en fonction du nombre d'éléments déjà présents.

Les tas sont donc particulièrement adaptés pour réaliser les files de priorité puisque la détermination du plus grand élément est immédiate et que la suppression de cet élément se fait avec une complexité logarithmique. Les tas sont également très utiles dans l'implémentation des algorithmes de tri car ils permettent d'atteindre une complexité algorithmique en `n×ln(n)`, ce qui est l'optimum.

> **Note :** En pratique, un tas est une forme d'arbre binaire équilibré dont la propriété récursive est que la racine de l'arbre est l'élément de plus grande valeur et que les deux branches de l'arbre sont eux-même des tas. La suppression de la racine, ainsi que l'ajout d'un nouvel élément, nécessite une réorganisation de l'arbre binaire, ce qui ne peut dépasser `ln(n)` opérations en raison de son aspect équilibré.
>
> Notez que les tas ne garantissent pas, contrairement aux B-arbres et aux arbres rouges et noirs, que tous les éléments situés à la gauche d'un noeud sont plus grands que le noeud lui-même et que tous les éléments situés à la droite sont plus petits. C'est pour cette raison qu'un tas n'est justement pas complètement trié, et que les algorithmes de gestion des tas ne font que conserver cet ordre partiel.
>
> La représentation des tas en mémoire peut être relativement difficile à comprendre. En général, il est d'usage de les stocker dans des tableaux, car les opérations de gestion des tas requièrent des itérateurs à accès aléatoires sur le conteneur sur lequel elles travaillent. Dans ce cas, les premiers éléments du tableau stockent les noeuds de l'arbre binaire du tas, et les feuilles sont placées dans la seconde moitié du tableau. Ainsi, un élément d'indice `i` a comme feuilles les éléments d'indice `2×i` et `2×i+1` (pour tout `i < n/2`). Reportez-vous à la bibliographie pour plus de renseignements sur les structures de données et les notions algorithmiques associées.

Les algorithmes de manipulation des tas sont déclarés comme suit dans l'en-tête `algorithm` :

```
template <class RandomAccessIterator>
void make_heap(RandomAccessIterator premier, RandomAccessIterator dernier);
```

```
template <class RandomAccessIterator, class Compare>
void make_heap(RandomAccessIterator premier, RandomAccessIterator dernier,
    Compare c);

template <class RandomAccessIterator>
void pop_heap(RandomAccessIterator premier, RandomAccessIterator dernier);

template <class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator premier, RandomAccessIterator dernier,
    Compare c);

template <class RandomAccessIterator>
void push_heap(RandomAccessIterator premier, RandomAccessIterator dernier);

template <class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator premier, RandomAccessIterator dernier,
    Compare c);

template <class RandomAccessIterator>
void sort_heap(RandomAccessIterator premier, RandomAccessIterator dernier);

template <class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator premier, RandomAccessIterator dernier,
    Compare c);
```

L'algorithme `make_heap` permet de construire un nouveau tas à partir d'une séquence d'éléments quelconque. Il prend simplement en paramètre les itérateurs de début et de fin de cette séquence, et ne retourne rien. Sa complexité est une fonction linéaire du nombre d'éléments référencés par ces deux itérateurs.

Les algorithmes `pop_heap` et `push_heap` permettent respectivement de supprimer la tête d'un tas existant et d'ajouter un nouvel élément dans un tas. `pop_heap` prend en paramètre deux itérateurs référençant le premier et le dernier élément du tas. Il place le premier élément du tas en dernière position et réorganise les éléments restants de telle sorte que les `dernier-premier-1` éléments constituent un nouveau tas. L'algorithme `push_heap` en revanche effectue le travaille inverse : il prend en paramètre deux itérateurs référençant une séquence dont les premiers éléments sauf le dernier constituent un tas et y ajoute l'élément référencé par l'itérateur `dernier-1`. Ces deux opérations effectuent leur travail avec une complexité logarithmique.

Enfin, l'algorithme `sort_heap` permet simplement de trier une séquence ayant la structure de tas. Sa complexité est `n×ln(n)`, où `n` est le nombre d'éléments de la séquence.

Exemple 18-19. Algorithmes de manipulation des tas

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    int t[10] = {5, 8, 1, 6, 7, 9, 4, 3, 0, 2};
    // Construit un tas à partir de ce tableau :
    make_heap(t, t+10);
    // Affiche le tas :
```

```
    int i;
    for (i=0; i<10; ++i)
        cout << t[i] << " ";
    cout << endl;
    // Supprime l'élément de tête :
    pop_heap(t, t+10);
    // L'élément de tête est en position 9 :
    cout << "Max = " << t[9] << endl;
    // Affiche le nouveau tas :
    for (i=0; i<9; ++i)
        cout << t[i] << " ";
    cout << endl;
    // Ajoute un élément :
    t[9] = 6;
    push_heap(t, t+10);
    // Affiche le nouveau tas :
    for (i=0; i<10; ++i)
        cout << t[i] << " ";
    cout << endl;
    // Tri le tas :
    sort_heap(t, t+10);
    // Affiche le tableau ainsi trié :
    for (i=0; i<10; ++i)
        cout << t[i] << " ";
    cout << endl;
    return 0;
}
```

18.3.2. Opérations de tri

Les opérations de tri de la bibliothèque standard s'appuient sur les algorithmes de manipulation des tas que l'on vient de voir. Ces méthodes permettent d'effectuer un tri total des éléments d'une séquence, un tri stable, légèrement moins performant que le précédent mais permettant de conserver l'ordre relatif des éléments équivalents, et un tri partiel.

Les algorithmes de tri sont déclarés comme suit dans l'en-tête `algorithm` :

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator premier, RandomAccessIterator dernier);

template <class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator premier, RandomAccessIterator dernier,
    Compare c);

template <class RandomAccessIterator>
void stable_sort(RandomAccessIterator premier, RandomAccessIterator dernier);

template <class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator premier, RandomAccessIterator dernier,
    Compare c);
```

Les algorithmes `sort` et `stable_sort` s'utilisent de la même manière et permettent de trier complètement la séquence qui leur est spécifiée à l'aide des deux itérateurs `premier` et `dernier`. Ces deux algorithmes effectuent un tri par ordre croissant en utilisant l'opérateur d'infériorité du type des

éléments de la séquence à trier. Cependant, il est également possible d'utiliser un autre critère, en spécifiant un foncteur binaire en troisième paramètre. Ce foncteur doit être capable de comparer deux éléments de la séquence à trier et d'indiquer si le premier est ou non le plus petit au sens de la relation d'ordre qu'il utilise.

Exemple 18-20. Algorithme de tri

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    int t[10] = {2, 3, 7, 5, 4, 1, 8, 0, 9, 6};
    // Trie le tableau :
    sort(t, t+10);
    // Affiche le résultat :
    int i;
    for (i=0; i<10; ++i)
        cout << t[i] << " ";
    cout << endl;
    return 0;
}
```

Il se peut que plusieurs éléments de la séquence soient considérés comme équivalents par la relation d'ordre utilisée. Par exemple, il est possible de trier des structures selon l'un de leurs champs, et plusieurs éléments peuvent avoir la même valeur dans ce champ sans pour autant être strictement égaux. Dans ce cas, il peut être nécessaire de conserver l'ordre relatif initial de ces éléments dans la séquence à trier. L'algorithme `sort` ne permet pas de le faire, cependant, l'algorithme `stable_sort` garantit la conservation de cet ordre relatif, au prix d'une complexité algorithmique légèrement supérieure. En effet, la complexité de `stable_sort` est $n \times \ln^2(n)$ (où `n` est le nombre d'éléments à trier), alors que celle de l'algorithme `sort` n'est que de $n \times \ln(n)$. Hormis cette petite différence, les deux algorithmes sont strictement équivalents.

Dans certaines situations, il n'est pas nécessaire d'effectuer un tri total des éléments. En effet, le tri des premiers éléments d'une séquence seulement ou bien seule la détermination du nième élément d'un ensemble peuvent être désirés. À cet effet, la bibliothèque standard fournit les algorithmes suivants :

```
template <class RandomAccessIterator>
void partial_sort(RandomAccessIterator premier,
    RandomAccessIterator pivot, RandomAccessIterator dernier);

template <class InputIterator, class RandomAccessIterator>
RandomAccessIterator partial_sort_copy(
    InputIterator premier, InputIterator dernier,
    RandomAccessIterator debut_resultat, RandomAccessIterator fin_resultat);

template <class RandomAccessIterator, class Compare>
void partial_sort(
    RandomAccessIterator premier, RandomAccessIterator fin_tri,
    RandomAccessIterator dernier, Compare c);

template <class InputIterator, class RandomAccessIterator,
    class Compare>
RandomAccessIterator partial_sort_copy(
```

```
    InputIterator premier, InputIterator dernier,
    RandomAccessIterator debut_resultat, RandomAccessIterator fin_resultat,
    Compare c);

template <class RandomAccessIterator>
void nth_element(RandomAccessIterator premier, RandomAccessIterator position,
    RandomAccessIterator dernier);

template <class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator premier, RandomAccessIterator position,
    RandomAccessIterator dernier,  Compare c);
```

L'algorithme `partial_sort` permet de n'effectuer qu'un tri partiel d'une séquence. Cet algorithme peut être utilisé lorsqu'on désire n'obtenir que les premiers éléments de la séquence triée. Cet algorithme existe en deux versions. La première version prend en paramètre l'itérateur de début de la séquence, l'itérateur de la position du dernier élément de la séquence qui sera triée à la fin de l'exécution de l'algorithme, et l'itérateur de fin de la séquence. La deuxième version, nommée `partial_sort_copy`, permet de copier le résultat du tri partiel à un autre emplacement que celui de la séquence initiale. Cette version de l'algorithme de tri partiel prend alors deux couples d'itérateurs en paramètre, le premier spécifiant la séquence sur laquelle l'algorithme doit travailler et le deuxième l'emplacement destination dans lequel le résultat doit être stocké. Enfin, comme pour tous les autres algorithmes, il est possible de spécifier un autre opérateur de comparaison que l'opérateur d'infériorité utilisé par défaut en fournissant un foncteur binaire en dernier paramètre.

Exemple 18-21. Algorithme de tri partiel

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    int t[10] = {2, 3, 7, 5, 4, 1, 8, 0, 9, 6};
    // Trie les 5 premiers éléments du tableau :
    partial_sort(t, t+5, t+10);
    // Affiche le résultat :
    int i;
    for (i=0; i<10; ++i)
        cout << t[i] << " ";
    cout << endl;
    return 0;
}
```

La complexité de l'algorithme `partial_sort` est `n×ln(m)`, où `n` est la taille de la séquence sur laquelle l'algorithme travaille et `m` est le nombre d'éléments triés à obtenir.

L'algorithme `nth_element` permet quant à lui de calculer la valeur d'un élément de rang donné dans le conteneur si celui-ci était complètement trié. Cet algorithme prend en paramètre l'itérateur de début de la séquence à traiter, l'itérateur référençant l'emplacement qui recevra l'élément qui sera placé à sa position à la fin de l'opération de tri partiel et l'itérateur de fin de la séquence. Il est également possible, comme pour les autres algorithmes, de spécifier un foncteur à utiliser pour tester l'infériorité des éléments de la séquence. À l'issue de l'appel, le n-ième élément de la séquence sera le même élément que celui qui se trouverait à cette position si la séquence était complètement triée

selon la relation d'ordre induite par l'opérateur d'infériorité ou par le foncteur fourni en paramètre. La complexité de l'algorithme `nth_element` est linéaire en fonction du nombre d'éléments de la séquence à traiter.

Exemple 18-22. Algorithme de positionnement du nième élément

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    int t[10] = {2, 3, 9, 6, 7, 5, 4, 0, 1, 8};
    // Trie tous les éléments un à un :
    int i;
    for (i=0; i<10; ++i)
    {
        nth_element(t, t+i, t+10);
        cout << "L'élément " << i <<
            " a pour valeur " << t[i] << endl;
    }
    return 0;
}
```

Enfin, et bien que ces algorithmes ne fassent pas à proprement parler des opérations de tri, la bibliothèque standard fournit deux algorithmes permettant d'obtenir le plus petit et le plus grand des éléments d'une séquence. Ces algorithmes sont déclarés de la manière suivante dans l'en-tête `algorithm` :

```
template <class ForwardIterator>
ForwardIterator min_element(ForwardIterator premier, ForwardIterator dernier);

template <class ForwardIterator, class Compare>
ForwardIterator min_element(ForwardIterator premier, ForwardIterator dernier,
    Compare c);

template <class ForwardIterator>
ForwardIterator max_element(ForwardIterator premier, ForwardIterator dernier);

template <class ForwardIterator, class Compare>
ForwardIterator max_element(ForwardIterator premier, ForwardIterator dernier,
    Compare c);
```

Ces deux algorithmes prennent en paramètre deux itérateurs permettant de définir la séquence des éléments dont le minimum et le maximum doivent être déterminés. Ils retournent un itérateur référençant respectivement le plus petit et le plus grand des éléments de cette séquence. La complexité de ces algorithmes est proportionnelle à la taille de la séquence fournie en paramètre.

Exemple 18-23. Algorithmes de détermination du maximum et du minimum

```
#include <iostream>
#include <algorithm>

using namespace std;
```

```
int main(void)
{
    int t[10] = {5, 2, 4, 6, 3, 7, 9, 1, 0, 8};
    // Affiche le minimum et le maximum :
    cout << *min_element(t, t+10) << endl;
    cout << *max_element(t, t+10) << endl;
    return 0;
}
```

18.3.3. Opérations de recherche binaire

Les opérations de recherche binaire de la bibliothèque standard sont des opérations qui permettent de manipuler des séquences d'éléments déjà triées en se basant sur cet ordre. Les principales fonctionnalités de ces algorithmes sont de rechercher les positions des éléments dans ces séquences en fonction de leur valeur.

Les principaux algorithmes de recherche binaire sont les algorithmes `lower_bound` et `upper_bound`. Ces algorithmes sont déclarés comme suit dans l'en-tête `algorithm` :

```
template <class ForwardIterator, class T>
ForwardIterator lower_bound(ForwardIterator premier, ForwardIterator dernier,
    const T &valeur);

template <class ForwardIterator, class T, class Compare>
ForwardIterator lower_bound(ForwardIterator premier, ForwardIterator dernier,
    const T &valeur, Compare c);

template <class ForwardIterator, class T>
ForwardIterator upper_bound(ForwardIterator premier, ForwardIterator dernier,
    const T &valeur);

template <class ForwardIterator, class T, class Compare>
ForwardIterator upper_bound(ForwardIterator premier, ForwardIterator dernier,
    const T &valeur, Compare c);
```

L'algorithme `lower_bound` détermine la première position à laquelle la valeur `valeur` peut être insérée dans la séquence ordonnée spécifiée par les itérateurs `premier` et `dernier` sans en briser l'ordre. De même, l'algorithme `upper_bound` détermine la dernière position à laquelle la valeur `valeur` peut être insérée sans casser l'ordre de la séquence sur laquelle il travaille. Il est supposé ici que l'insertion se ferait avant les éléments indiqués par ces itérateurs, comme c'est généralement le cas pour tous les conteneurs.

Si le programmeur veut déterminer simultanément les deux itérateurs renvoyés par les algorithmes `lower_bound` et `upper_bound`, il peut utiliser l'algorithme `equal_range` suivant :

```
template <class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator premier, ForwardIterator dernier,
        const T &valeur);

template <class ForwardIterator, class T, class Compare>
pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator premier, ForwardIterator dernier,
```

```
        const T &valeur, Compare comp);
```

Cet algorithme renvoie une paire d'itérateurs contenant respectivement la première et la dernière des positions auxquelles la valeur `valeur` peut être insérée sans perturber l'ordre de la séquence identifiée par les itérateurs `premier` et `dernier`.

Exemple 18-24. Algorithmes de détermination des bornes inférieures et supérieures

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    int t[10] = {1, 2, 4, 4, 4, 5, 8, 9, 15, 20};
    // Détermine les positions possibles d'insertion
    // d'un 4 :
    cout << "4 peut être inséré de " <<
        lower_bound(t, t+10, 4) - t <<
        " à " <<
        upper_bound(t, t+10, 4) - t << endl;
    // Récupère ces positions directement
    // avec equal_range :
    pair<int *, int *> p = equal_range(t, t+10, 4);
    cout << "Equal range donne l'intervalle [" <<
        p.first-t << ", " << p.second-t << "]";
    cout << endl;
    return 0;
}
```

Comme pour la plupart des algorithmes de la bibliothèque standard, il est possible de spécifier un foncteur qui devra être utilisé par les algorithmes de recherche binaire dans les comparaisons d'infériorité des éléments de la séquence.

Enfin, l'algorithme `binary_search` permet de déterminer si un élément d'un conteneur au moins est équivalent à une valeur donnée au sens de l'opérateur d'infériorité ou au sens d'un foncteur fourni en paramètre. Cet algorithme est déclaré de la manière suivante dans l'en-tête `algorithm` :

```
template <class ForwardIterator, class T>
bool binary_search(ForwardIterator premier, ForwardIterator dernier,
    const T &valeur);

template <class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator premier, ForwardIterator dernier,
    const T &valeur, Compare c);
```

Cet algorithme prend en paramètre les deux itérateurs définissant la séquence d'éléments à tester, la valeur avec laquelle ses éléments doivent être testés, et éventuellement un foncteur permettant de réaliser une opération de comparaison autre que celle de l'opérateur d'infériorité. Il renvoie un booléen indiquant si un des éléments au moins du conteneur est équivalent à la valeur fournie en paramètre.

Note : La relation d'équivalence utilisée par cet algorithme n'est pas celle induite par l'opérateur d'égalité des éléments. En réalité, deux éléments `x` et `y` sont considérés comme équivalents si et seulement si les deux inéquations `x<y` et `y<x` sont fausses. C'est la raison pour laquelle le foncteur fourni en paramètre ne doit pas définir la relation d'égalité, mais la relation d'infériorité.

Cette distinction a son importance si certains éléments de la séquence ne sont pas comparables ou si l'opérateur d'égalité définit une autre relation que l'opérateur d'infériorité. Bien entendu, en pratique, ces deux inéquations signifie souvent que les valeurs `x` et `y` sont égales.

Exemple 18-25. Algorithme de recherche binaire

```
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;

struct A
{
    int numero;   // Numéro unique de l'élément
    string nom;   // Nom de l'élément

    A(const char *s) :
        nom(s)
    {
        // Affecte un nouveau numéro :
        static int i=0;
        numero = ++i;
    }

    // Opérateur de classement :
    bool operator<(const A &a) const
    {
        return (numero < a.numero);
    }

    // Opérateur d'égalité (jamais utilisé) :
    bool operator==(const A &a) const
    {
        return (nom == a.nom);
    }
};

int main(void)
{
    // Construit un tableau d'éléments triés
    // (par construction, puisque le numéro est incrémenté
    // à chaque nouvel objet) :
    A t[5] = {"Jean", "Marc", "Alain", "Ariane", "Sophie"};
    // Cette instance a le même nom que t[1]
    // mais ne sera pas trouvé car son numéro est différent :
    A test("Marc");
    // Effectue la recherche de test dans le tableau :
    if (binary_search(t, t+5, test))
    {
        cout << "(" << test.numero << ", " <<
```

```
            test.nom << ") a été trouvé" << endl;
    }
    else
    {
        cout << "(" << test.numero << ", " <<
            test.nom << ") n'a pas été trouvé" << endl;
    }
    return 0;
}
```

La complexité algorithmique de tous ces algorithmes est logarithmique en fonction du nombre d'éléments de la séquence sur laquelle ils travaillent. Ils s'appuient sur le fait que cette séquence est déjà triée pour atteindre cet objectif.

18.4. Opérations de comparaison

Afin de faciliter la comparaison de conteneurs de natures différentes pour lesquels, de surcroît, il n'existe pas forcément d'opérateurs de comparaison, la bibliothèque standard fournit plusieurs algorithmes de comparaison. Ces algorithmes sont capables d'effectuer une comparaison élément à élément des différents conteneurs pour vérifier leur égalité en terme d'éléments contenus, ou de déterminer une relation d'ordre au sens lexicographique. Enfin, il est possible de déterminer les éléments par lesquels deux conteneurs se différencient.

L'algorithme général de comparaison des conteneurs est l'algorithme `equal`. Cet algorithme est déclaré comme suit dans l'en-tête `algorithm` :

```
template <class InputIterator1, class InputIterator2>
bool equal(InputIterator1 premier1, InputIterator1 dernier1,
    InputIterator2 premier2);

template <class InputIterator1, class InputIterator2, class BinaryPredicate>
bool equal(InputIterator1 premier1, InputIterator1 dernier1,
    InputIterator2 premier2, BinaryPredicate p);
```

Comme vous pouvez le constater d'après cette déclaration, l'algorithme `equal` prend en paramètre un couple d'itérateurs décrivant la séquence d'éléments qui doivent être pris en compte dans la comparaison ainsi qu'un itérateur sur le premier élément du deuxième conteneur. Les éléments référencés successivement par les itérateurs `premier1` et `premier2` sont ainsi comparés, jusqu'à ce qu'une différence soit détectée ou que l'itérateur `dernier1` du premier conteneur soit atteint. La valeur retournée est `true` si les deux séquences d'éléments des deux conteneurs sont égales élément à élément, et `false` sinon. Bien entendu, il est possible de spécifier un foncteur binaire que l'algorithme devra utiliser pour réaliser les comparaisons entre les éléments des deux conteneurs. S'il est spécifié, ce foncteur est utilisé pour déterminer si les éléments comparés sont égaux ou non.

> **Note :** Notez bien ici que le foncteur fourni permet de tester l'égalité de deux éléments et non l'infériorité, comme c'est le cas avec la plupart des autres algorithmes.

S'il s'avère que les deux conteneurs ne sont pas égaux membre à membre, il peut être utile de déterminer les itérateurs des deux éléments qui ont fait échouer le test d'égalité. Cela peut être réalisé à l'aide de l'algorithme `mismatch` dont on trouvera la déclaration dans l'en-tête `algorithm` :

```
template <class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 premier1, InputIterator1 dernier1,
        InputIterator2 premier2);

template <class InputIterator1, class InputIterator2, class BinaryPredicate>
pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 premier1, InputIterator1 dernier1,
        InputIterator2 premier2, BinaryPredicate p);
```

Cet algorithme fonctionne exactement de la même manière que l'algorithme `equal`. Cependant, contrairement à ce dernier, sa valeur de retour est une paire d'itérateurs des deux conteneurs référençant les éléments respectifs qui ne sont pas égaux au sens de l'opération de comparaison utilisée par l'algorithme (que ce soit l'opérateur d'égalité ou le foncteur fourni en paramètre). Si les deux conteneurs sont effectivement égaux, la valeur retournée est la paire contenant l'itérateur `dernier1` et l'itérateur correspondant dans le deuxième conteneur.

Exemple 18-26. Algorithme de comparaison de conteneurs

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    int t1[10] = {5, 6, 4, 7, 8, 9, 2, 1, 3, 0};
    int t2[10] = {5, 6, 4, 7, 9, 2, 1, 8, 3, 0};
    // Compare les deux tableaux :
    if (!equal(t1, t1+10, t2))
    {
        // Détermine les éléments différents :
        pair<int *, int *> p =
            mismatch(t1, t1+10, t2);
        cout << *p.first << " est différent de " <<
            *p.second << endl;
    }
    return 0;
}
```

Enfin, la bibliothèque standard fournit un algorithme de comparaison général permettant de déterminer si un conteneur est inférieur à un autre conteneur selon la relation d'ordre lexicographique induite par l'opérateur d'infériorité du type de leurs éléments. Rappelons que l'ordre lexicographique est celui utilisé par le dictionnaire : les éléments sont examinés un à un et dans leur ordre d'apparition et la comparaison s'arrête dès que deux éléments différents sont trouvés. En cas d'égalité totale, le plus petit des conteneurs est celui qui contient le moins d'éléments.

L'algorithme de comparaison lexicographique est l'algorithme `lexicographical_compare`. Il est déclaré comme suit dans l'en-tête `algorithm` :

```
template <class InputIterator1, class InputIterator2>
bool lexicographical_compare(InputIterator1 premier1, InputIterator1 dernier1,
    InputIterator2 premier2, InputIterator2 dernier2);

template <class InputIterator1, class InputIterator2, class Compare>
```

```
bool lexicographical_compare(InputIterator1 premier1, InputIterator1 dernier1,
    InputIterator2 premier2, InputIterator2 dernier2,
    Compare c);
```

Cet algorithme prend en paramètre deux couples d'itérateurs grâce auxquels le programmeur peut spécifier les deux séquences d'éléments à comparer selon l'ordre lexicographique. Comme à l'accoutumée, il est également possible de fournir un foncteur à utiliser pour les tests d'infériorité entre les éléments des deux conteneurs. La valeur retournée par l'algorithme `lexicographical_compare` est `true` si le premier conteneur est strictement plus petit que le deuxième et `false` sinon.

Exemple 18-27. Algorithme de comparaison lexicographique

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    int t1[10] = {5, 6, 4, 7, 8, 9, 2, 1, 3, 0};
    int t2[10] = {5, 6, 4, 7, 9, 2, 1, 8, 3, 0};
    // Compare les deux tableaux :
    if (lexicographical_compare(t1, t1+10, t2, t2+10))
    {
        cout << "t1 est plus petit que t2" << endl;
    }
    return 0;
}
```

Tous ces algorithmes de comparaison s'exécutent avec une complexité linéaire en fonction du nombre d'éléments à comparer.

18.5. Opérations ensemblistes

En mathématiques, il est possible d'effectuer différents types d'opérations sur les ensembles. Ces opérations comprennent la détermination de l'inclusion d'un ensemble dans un autre, leur union (c'est-à-dire le regroupement de tous leurs éléments), leur intersection (la sélection de leurs éléments communs), leur différence (la suppression des éléments d'un ensemble qui appartiennent aussi à un autre ensemble) et leur partitionnement (le découpage d'un ensemble en sous-ensemble dont les éléments vérifient une propriété discriminante).

La bibliothèque standard fournit tout un ensemble d'algorithmes qui permettent d'effectuer les opérations ensemblistes classiques sur les conteneurs triés. Tous ces algorithmes sont décrits ci-dessous et sont classés selon la nature des opérations qu'ils réalisent.

> **Note :** Remarquez ici que la notion de tri est importante : les algorithmes s'appuient sur cette propriété des conteneurs pour effectuer leur travail. En contrepartie de cette contrainte, les performances de ces algorithmes sont excellentes.

18.5.1. Opérations d'inclusion

L'inclusion d'un ensemble dans un autre peut être réalisée à l'aide de l'algorithme `includes`. Cet algorithme est déclaré comme suit dans l'en-tête `algorithm` :

```
template <class InputIterator1, class InputIterator2>
bool includes(InputIterator1 premier1, InputIterator1 dernier1,
    InputIterator2 premier2, InputIterator2 dernier2);

template <class InputIterator1, class InputIterator2, class Compare>
bool includes(InputIterator1 premier1, InputIterator1 dernier1,
    InputIterator2 premier2, InputIterator2 dernier2, Compare c);
```

L'algorithme `includes` prend en paramètre deux couples d'itérateurs permettant de définir les séquences d'éléments des deux ensembles sur lesquels il doit travailler. La valeur retournée par cet algorithme est `true` si tous les éléments de la séquence identifiée par les itérateurs `premier2` et `dernier2` sont également présents dans la séquence identifiée par les itérateurs `premier1` et `dernier1`. L'algorithme considère qu'un élément est présent dans un ensemble s'il existe au moins un élément de cet ensemble qui lui est identique. Chaque élément utilisé de l'ensemble ne l'est qu'une seule fois, ainsi, si l'ensemble dont on teste l'inclusion dispose de plusieurs copies du même élément, il faut qu'il y en ait autant dans l'ensemble conteneur pour que le test d'inclusion soit valide.

Bien entendu, il est possible d'utiliser une autre relation que l'égalité pour déterminer l'appartenance d'un élément à un ensemble, pour cela, il suffit de fournir un foncteur binaire en dernier paramètre. Ce prédicat doit prendre deux éléments en paramètre et renvoyer `true` si le premier élément est inférieur au second, et `false` dans le cas contraire.

> **Note :** Il est important que le foncteur d'infériorité spécifié soit compatible avec la relation d'ordre utilisée pour le tri des éléments des conteneurs. Si ce n'est pas le cas, l'algorithme peut ne pas fonctionner correctement.

Exemple 18-28. Algorithme de détermination d'inclusion

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    int t1[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int t2[3] = {4, 5, 6};
    if (includes(t1, t1+10, t2, t2+3))
        cout << "t1 contient t2" << endl;
    return 0;
}
```

La complexité de l'algorithme `includes` est `n+m`, où `n` et `m` sont respectivement les tailles des deux conteneurs qui lui sont fournis en paramètre.

18.5.2. Opérations d'intersection

L'intersection de deux ensembles peut être réalisée à l'aide de l'algorithme `set_intersection`. Cet algorithme est déclaré de la manière suivante dans l'en-tête `algorithm` :

```
template <class InputIterator1, class InputIterator2,
    class OutputIterator>
OutputIterator set_intersection(InputIterator1 premier1, InputIterator1 dernier1,
    InputIterator2 premier2, InputIterator2 dernier2,
    OutputIterator destination);

template <class InputIterator1, class InputIterator2,
    class OutputIterator, class Compare>
OutputIterator set_intersection(InputIterator1 premier1, InputIterator1 dernier1,
    InputIterator2 premier2, InputIterator2 dernier2,
    OutputIterator destination, Compare c);
```

Cet algorithme prend en paramètre les itérateurs de début et de fin des deux conteneurs dont l'intersection doit être déterminée, ainsi qu'un itérateur référençant l'emplacement destination où les éléments de l'intersection doivent être stockés. Pour ceux qui le désirent, il est également possible de spécifier un foncteur que l'algorithme utilisera pour effectuer les comparaisons d'infériorité entre les éléments des deux conteneurs fournis en paramètre. Ce foncteur devra bien entendu être compatible avec la relation d'ordre selon laquelle les conteneurs passés en paramètre sont triés.

L'algorithme copie à l'emplacement destination tous les éléments du premier conteneur qui font également partie du deuxième. Le critère d'appartenance à un ensemble est, comme pour l'algorithme `includes`, le fait qu'il existe au moins un élément dans le deuxième ensemble égal à l'élément considéré. De même, si plusieurs copies d'un même élément se trouvent dans chaque ensemble, le nombre de copies de l'intersection sera le plus petit nombre de copies de l'élément dans les deux ensembles sources.

Exemple 18-29. Algorithme d'intersection d'ensembles

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    int t1[10] = {2, 4, 6, 8, 9, 10, 15, 15, 15, 17};
    int t2[10] = {1, 4, 5, 8, 11, 15, 15, 16, 18, 19};
    int t[10];
    // Effectue l'intersection de t1 et de t2 :
    int *fin = set_intersection(t1, t1+10, t2, t2+10, t);
    // Affiche le résultat :
    int *p = t;
    while (p != fin)
    {
        cout << *p << " ";
        ++p;
    }
    cout << endl;
    return 0;
}
```

La complexité de l'algorithme est `n+m`, où `n` et `m` sont respectivement les tailles des deux conteneurs qui lui sont fournis en paramètre.

18.5.3. Opérations d'union et de fusion

La bibliothèque standard fournit plusieurs algorithmes permettant de réaliser l'union de deux ensembles. Ces variantes se distinguent par la manière qu'elles ont de traiter le cas des éléments en multiples exemplaires.

L'algorithme `set_union` considère que les éléments équivalents des deux ensembles sont les mêmes entités et ne les place qu'une seule fois dans l'ensemble résultat de l'union. Toutefois, si ces éléments sont en plusieurs exemplaires dans un des ensembles source, ils apparaîtront également en plusieurs exemplaires dans le résultat. Autrement dit, le nombre d'éléments présents dans l'ensemble destination est le nombre maximum du compte de ses occurrences dans chacun des deux ensembles source.

Inversement, l'algorithme `merge` effectue une union au sens large et ajoute les éléments de chaque ensemble dans l'ensemble résultat sans considérer leurs valeurs. Ainsi, le nombre d'éléments du résultat est strictement égal à la somme des nombres des éléments de chaque conteneur source.

Afin de distinguer ces deux comportements, on peut dire que l'algorithme `set_union` réalise l'*union* des deux ensembles, alors que l'algorithme `merge` réalise leur *fusion*.

Tous ces algorithmes sont déclarés comme suit dans l'en-tête `algorithm` :

```
template <class InputIterator1, class InputIterator2,
    class OutputIterator>
OutputIterator set_union(InputIterator1 premier1, InputIterator1 dernier1,
    InputIterator2 premier2, InputIterator2 dernier2,
    OutputIterator destination);

template <class InputIterator1, class InputIterator2,
    class OutputIterator, class Compare>
OutputIterator set_union(InputIterator1 premier1, InputIterator1 dernier1,
    InputIterator2 premier2, InputIterator2 dernier2,
    OutputIterator destination, Compare c);

template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator merge(InputIterator1 premier1, InputIterator1 dernier1,
    InputIterator2 premier2, InputIterator2 dernier2,
    OutputIterator destination);

template <class InputIterator1, class InputIterator2,
    class OutputIterator, class Compare>
OutputIterator merge(InputIterator1 premier1, InputIterator1 dernier1,
    InputIterator2 dernier2, InputIterator2 premier2,
    OutputIterator destination, Compare c);
```

Comme vous pouvez le constater, ils prennent tous en paramètre les itérateurs permettant de spécifier les deux ensembles ainsi qu'un itérateur `destination` indiquant l'emplacement où les éléments de l'union ou de la fusion doivent être stockés. Enfin, si le programmeur le désire, il peut également donner le foncteur définissant la relation d'ordre selon laquelle les ensembles sont triés.

Exemple 18-30. Algorithmes d'union et de fusion d'ensembles

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    int t1[4] = {1, 2, 5, 5};
    int t2[6] = {3, 4, 5, 5, 5, 7};
    int t[10];
    // Effectue l'union de t1 et de t2 :
    int *fin = set_union(t1, t1+4, t2, t2+6, t);
    // Affiche le résultat :
    int *p = t;
    while (p != fin)
    {
        cout << *p << " ";
        ++p;
    }
    cout << endl;
    // Effectue la fusion de t1 et de t2 :
    fin = merge(t1, t1+4, t2, t2+6, t);
    // Affiche le résultat :
    p = t;
    while (p != fin)
    {
        cout << *p << " ";
        ++p;
    }
    cout << endl;
    return 0;
}
```

La bibliothèque standard fournit également une version modifiée de l'algorithme `merge` dont le but est de fusionner deux parties d'une même séquence d'éléments triées indépendamment l'une de l'autre. Cet algorithme permet d'effectuer la fusion sur place, et ne travaille donc que sur un seul conteneur. Il s'agit de l'algorithme `inplace_merge`, qui est déclaré comme suit :

```
template <class BidirectionalIterator>
void inplace_merge(BidirectionalIterator premier,
    BidirectionalIterator separation,
    BidirectionalIterator dernier);

template <class BidirectionalIterator, class Compare>
void inplace_merge(BidirectionalIterator premier,
    BidirectionalIterator separation,
    BidirectionalIterator dernier, Compare c);
```

Cet algorithme effectue la fusion des deux ensembles identifiés respectivement par les itérateurs `premier` et `separation` d'une part, et par les itérateurs `separation` et `dernier` d'autre part. Enfin, si besoin est, il est possible de spécifier le foncteur selon lequel ces deux ensembles sont triés.

Exemple 18-31. Algorithme de réunification de deux sous-ensembles

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    int t[10] = {1, 5, 9, 0, 2, 3, 4, 6, 7, 8};
    // Fusionne les deux sous-ensembles de t
    // (la séparation est au troisième élément) :
    inplace_merge(t, t+3, t+10);
    // Affiche le résultat :
    int i;
    for (i=0; i<10; ++i)
    {
        cout << t[i] << " ";
    }
    cout << endl;
    return 0;
}
```

Tous les algorithmes d'union et de fusion ont une complexité `n+m`, où `n` et `m` sont les tailles des deux ensembles à fusionner ou à réunir.

18.5.4. Opérations de différence

La différence entre deux ensembles peut être réalisée avec l'algorithme `set_difference`. Cet algorithme supprime du premier ensemble tous les éléments du second, si nécessaire. Chaque élément n'est supprimé qu'une seule fois, ainsi, si le premier ensemble contient plusieurs éléments identiques et que le deuxième ensemble en contient moins, les éléments résiduels après suppression seront présents dans la différence.

La bibliothèque standard fournit également un algorithme de suppression symétrique, l'algorithme `set_symmetric_difference`, qui construit un nouvel ensemble contenant tous les éléments des deux ensembles qui ne se trouvent pas dans l'autre. Il s'agit en fait de l'union des deux différences des deux ensembles.

> **Note :** Remarquez que le mot « symmetric » s'écrit avec deux '`m`' en anglais. Ne vous étonnez donc pas d'obtenir des erreurs de compilation si vous écrivez `set_symmetric_difference` à la française !

Les algorithmes `set_difference` et `set_symmetric_difference` sont déclarés comme suit dans l'en-tête `algorithm` :

```
template <class InputIterator1, class InputIterator2,
    class OutputIterator>
OutputIterator set_difference(
    InputIterator1 premier1, InputIterator1 dernier1,
    InputIterator2 premier2, InputIterator2 dernier2,
    OutputIterator destination);

template <class InputIterator1, class InputIterator2,
```

```
    class OutputIterator, class Compare>
OutputIterator set_difference(
    InputIterator1 premier1, InputIterator1 dernier1,
    InputIterator2 premier2, InputIterator2 dernier2,
    OutputIterator destination, Compare c);

template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_symmetric_difference(
    InputIterator1 premier, InputIterator1 dernier,
    InputIterator2 premier, InputIterator2 dernier2,
    OutputIterator destination);

template <class InputIterator1, class InputIterator2,
    class OutputIterator, class Compare>
OutputIterator set_symmetric_difference(
    InputIterator1 premier1, InputIterator1 dernier1,
    InputIterator2 premier2, InputIterator2 dernier2,
    OutputIterator destination, Compare c);
```

Ils prennent tous deux paires d'itérateurs identifiant les deux ensembles dont la différence doit être calculée ainsi qu'un itérateur référençant l'emplacement destination dans lequel le résultat doit être placé. Comme à l'accoutumée, il est possible d'indiquer le foncteur permettant à l'algorithme de réaliser les tests d'infériorité entre deux éléments et selon lequel les ensembles sont triés. La complexité de ces algorithmes est `n+m`, où `n` et `m` sont les nombres d'éléments des deux ensembles sur lesquels les algorithmes opèrent.

Exemple 18-32. Algorithmes de différence d'ensembles

```
#include <iostream>
#include <algorithm>

using namespace std;

int main(void)
{
    int t1[10] = {0, 1, 5, 7, 7, 7, 8, 8, 9, 10};
    int t2[10] = {0, 2, 3, 7, 9, 11, 12, 12, 13, 14};
    int t[20];
    // Calcule la différence de t1 et de t2 :
    int *fin = set_difference(t1, t1+10, t2, t2+10, t);
    // Affiche le résultat :
    int *p = t;
    while (p != fin)
    {
        cout << *p << " ";
        ++p;
    }
    cout << endl;
    // Calcule la différence symétrique de t1 et t2 :
    fin = set_symmetric_difference(t1, t1+10, t2, t2+10, t);
    // Affiche le résultat :
    int *p = t;
    while (p != fin)
    {
        cout << *p << " ";
```

```
        ++p;
    }
    cout << endl;
    // Calcule la différence symétrique de t1 et t2 :
    fin = set_symmetric_difference(t1, t1+10, t2, t2+10, t);
    // Affiche le résultat :
    p = t;
    while (p != fin)
    {
        cout << *p << " ";
        ++p;
    }
    cout << endl;
    return 0;
}
```

18.5.5. Opérations de partitionnement

L'algorithme `partition` de la bibliothèque standard permet de séparer les éléments d'un ensemble en deux sous-ensembles selon un critère donné. Les éléments vérifiant ce critère sont placés en tête de l'ensemble, et les éléments qui ne le vérifient pas sont placés à la fin. Cet algorithme est déclaré comme suit dans l'en-tête `algorithm` :

```
template <class ForwardIterator, class Predicate>
ForwardIterator partition(ForwardIterator premier,
    ForwardIterator dernier, Predicate p);
```

Les paramètres qui doivent être fournis à cet algorithme sont les itérateurs référençant le premier et le dernier élément de l'ensemble à partitionner, ainsi qu'un foncteur unaire permettant de déterminer si un élément vérifie le critère de partitionnement ou non. La valeur retournée est la position de la séparation entre les deux sous-ensembles générés par l'opération de partition.

Exemple 18-33. Algorithme de partitionnement

```
#include <iostream>
#include <functional>
#include <algorithm>

using namespace std;

bool parity_even(int i)
{
    return (i & 1) == 0;
}

int main(void)
{
    int t[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    // Partitionne le tableau en nombre pairs
    // et nombre impairs :
    partition(t, t+10, ptr_fun(&parity_even));
    // Affiche le résultat :
    int i;
```

```
    for (i=0; i<10; ++i)
        cout << t[i] << " ";
    cout << endl;
    return 0;
}
```

La complexité de l'algorithme `partition` est linéaire en fonction du nombre d'éléments de l'ensemble à partitionner. Cependant, l'opération de partitionnement n'est pas stable, c'est-à-dire que l'ordre relatif des éléments de même valeur et sur lesquels le prédicat du critère de partitionnement donne le même résultat n'est pas conservé. La bibliothèque standard fournit donc un autre algorithme, stable celui-là, mais qui s'exécute avec une complexité légèrement supérieure. Il s'agit de l'algorithme `stable_partition`, qui est déclaré comme suit dans l'en-tête `algorithm` :

```
template <class ForwardIterator, class Predicate>
ForwardIterator stable_partition(ForwardIterator premier,
    ForwardIterator dernier, Predicate p);
```

Comme vous pouvez le constater, cet algorithme s'utilise exactement de la même manière que l'algorithme `partition`. Toutefois, il garantit l'ordre relatif des éléments au sein des sous-ensembles générés par l'opération de partitionnement. La complexité de cet algorithme est `n` s'il dispose de suffisamment de mémoire, et `n×ln(n)` dans le cas contraire (`n` étant la taille de l'ensemble à partitionner).

Chapitre 19. Conclusion

Pour terminer, je rappellerai les principales règles pour réaliser de bons programmes. Sans organisation, aucun langage, aussi puissant soit-il, ne peut garantir le succès d'un projet. Voici donc quelques conseils :

- commentez votre code, mais ne tuez pas le commentaire en en mettant là où les opérations sont vraiment très simples ou décrites dans un document externe. Marquez les références aux documents externes dans les commentaires ;
- analysez le problème avant de commencer la programmation. Cela comprend plusieurs étapes. La première est de réfléchir aux structures de données à utiliser et aux opérations qu'on va leur appliquer (il faut donc identifier les classes). Il faut ensuite établir les relations entre les classes ainsi identifiées et leurs communications. Pour cela, on pourra faire des diagrammes d'événements qui identifient les différentes étapes du processus permettant de traiter une donnée. Enfin, on décrira chacune des méthodes des classes fonctionnellement, afin de savoir exactement quelles sont leurs entrées et les domaines de validité de celles-ci, leurs sorties, leurs effets de bords et les opérations effectuées. Enfin seulement on passera au codage. Si le codage implique de corriger les résultats des étapes précédentes, c'est que la conception a été incorrecte ou incomplète : il vaut mieux retourner en phase de conception un peu pour voir l'impact des modifications à faire. Cela permet de ne pas passer à coté d'un effet de bord inattendu, et donc d'éviter de perdre du temps dans la phase de mise au point ;
- ne considérez aucun projet, même un petit projet ou un projet personnel, comme un projet qui échappe à ces règles. Si vous devez interrompre le développement d'un projet pour une raison quelconque, vous serez content de retrouver le maximum d'informations sur lui. Il en est de même si vous désirez améliorer un ancien projet. Et si la conception a été bien faite, cette amélioration ne sera pas une verrue sur l'ancienne version du logiciel, contrairement à ce qui se passe trop souvent.

Voilà. Vous connaissez à présent la plupart des fonctionnalités du C++. J'espère que la lecture de ce cours vous aura été utile et agréable. Si vous voulez en savoir plus, consultez les Draft Papers, mais sachez qu'ils sont réellement difficiles à lire. Ils ne peuvent vraiment pas être pris pour un support de cours. Vous trouverez en annexe la description de l'organisation générale de ce document, plus quelques renseignements pour faciliter leur lecture.

Bonne continuation...

Annexe A. Liste des mots clés du C/C++

La liste des mots clés du C/C++ est donnée dans le tableau ci-dessous.

Tableau A-1. Mots clés du langange

Mots clés				
and	and_eq	asm	auto	bitand
bitor	bool	break	case	catch
char	class	compl	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	export
extern	false	float	for	friend
goto	if	inline	int	long
mutable	namespace	new	not	not_eq
operator	or	or_eq	private	protected
public	register	reinterpret_cast	return	short
signed	sizeof	static	static_cast	struct
switch	template	this	throw	true
try	typedef	typeid	typename	union
unsigned	using	virtual	void	volatile
wchar_t	while	xor	xor_eq	

Annexe B. Priorités des opérateurs

Cette annexe donne la priorité des opérateurs du langage C++, dans l'ordre décroissant. Cette priorité intervient dans l'analyse de toute expression et dans la détermination de son sens. Cependant, l'analyse des expressions peut être modifiée en changeant les priorités à l'aide de parenthèses.

Tableau B-1. Opérateurs du langage

Opérateur	Nom ou signification
`::`	Opérateur de résolution de portée
`[]`	Opérateur d'accès aux éléments de tableau
`()`	Opérateur d'appel de fonction
`type()`	Opérateur de transtypage explicite
`.`	Opérateur de sélection de membre
`->`	Opérateur de sélection de membre par déréférencement
`++`	Opérateur d'incrémentation post-fixe
`--`	Opérateur de décrémentation post-fixe
`new`	Opérateur de création dynamique d'objets
`new[]`	Opérateur de création dynamique de tableaux
`delete`	Opérateur de destruction des objets créés dynamiquement
`delete[]`	Opérateur de destruction des tableaux créés dynamiquement
`++`	Opérateur d'incrémentation préfixe
`--`	Opérateur de décrémentation préfixe
`*`	Opérateur de déréférencement
`&`	Opérateur d'adresse
`+`	Opérateur plus unaire
`-`	Opérateur négation unaire
`!`	Opérateur de négation logique
`~`	Opérateur de complément à un
`sizeof`	Opérateur de taille d'objet
`sizeof`	Opérateur de taille de type
`typeid`	Opérateur d'identification de type
`(type)`	Opérateur de transtypage
`const_cast`	Opérateur de transtypage de constance
`dynamic_cast`	Opérateur de transtypage dynamique
`reinterpret_cast`	Opérateur de réinterprétation
`static_cast`	Opérateur de transtypage statique
`.*`	Opérateur de sélection de membre par pointeur sur membre
`->*`	Opérateur de sélection de membre par pointeur sur membre par déréférencement
`*`	Opérateur de multiplication
`/`	Opérateur de division
`%`	Opérateur de reste de la division entière

Opérateur	Nom ou signification
`+`	Opérateur d'addition
`-`	Opérateur de soustraction
`<<`	Opérateur de décalage à gauche
`>>`	Opérateur de décalage à droite
`<`	Opérateur d'infériorité
`>`	Opérateur de supériorité
`<=`	Opérateur d'infériorité ou d'égalité
`>=`	Opérateur de supériorité ou d'égalité
`==`	Opérateur d'égalité
`!=`	Opérateur d'inégalité
`&`	Opérateur et binaire
`^`	Opérateur ou exclusif binaire
`|`	Opérateur ou inclusif binaire
`&&`	Opérateur et logique
`||`	Opérateur ou logique
`?:`	Opérateur ternaire
`=`	Opérateur d'affectation
`*=`	Opérateur de multiplication et d'affectation
`/=`	Opérateur de division et d'affectation
`%=`	Opérateur de modulo et d'affectation
`+=`	Opérateur d'addition et d'affectation
`-=`	Opérateur de soustraction et d'affectation
`<<=`	Opérateur de décalage à gauche et d'affectation
`>>=`	Opérateur de décalage à droite et d'affectation
`&=`	Opérateur de et binaire et d'affectation
`|=`	Opérateur de ou inclusif binaire et d'affectation
`^=`	Opérateur de ou exclusif binaire et d'affectation
`,`	Opérateur virgule

Annexe C. Draft Papers

Les Draft Papers sont vraiment une source d'informations très précise, mais ils ne sont pas vraiment structurés. En fait, ils ne sont destinés qu'aux éditeurs de logiciels désirant réaliser un compilateur, et la structure du document ressemble à un texte de loi (fortement technique en prime). Les exemples y sont rares, et quand il y en a, on ne sait pas à quel paragraphe ils se réfèrent. Enfin, nombre de termes non définis sont utilisés, et il faut lire le document pendant quelques 40 pages avant de commencer à le comprendre.

Afin de faciliter leur lecture, je donne ici quelques définitions, ainsi que la structure des Draft Papers.

Les Draft Papers sont constitués de deux grandes parties. La première traite du langage, de sa syntaxe et de sa sémantique. La deuxième partie décrit la bibliothèque standard C++.

La syntaxe est décrite dans la première partie de la manière BNF. Il vaut mieux être familiarisé avec cette forme de description pour la comprendre. Cela ne causera pas de problème cependant si l'on maîtrise déjà la syntaxe du C++.

Lors de la lecture de la deuxième partie, on ne s'attardera pas trop sur les fonctionnalités de gestion des langues et des jeux de caractères (locales). Elles ne sont pas nécessaires à la compréhension de la bibliothèque standard. Une fois les grands principes de la bibliothèque assimilés, les notions de locale pourront être approfondies.

Les termes suivants sont souvent utilisés et non définis (ou définis au milieu du document d'une manière peu claire). Leurs définitions pourront être d'un grand secours lors de lecture de la première partie des Draft Papers :

- *cv*, *cv qualified* : l'abréviation *cv* signifie ici *const* ou *volatile*. Ce sont donc les propriétés de constance et de volatilité ;
- un *agrégat* est un tableau ou une classe qui n'a pas de constructeurs, pas de fonctions virtuelles, et pas de donnée non statique *private* ou *protected* ;
- *POD* : cette abréviation signifie *plain ol' data*, ce qui n'est pas compréhensible a priori. En fait, un type *POD* est un type relativement simple, pour lequel aucun traitement particulier n'est nécessaire (pas de constructeur, pas de virtualité, etc.). La définition des types *POD* est récursive : une structure ou une union est un type *POD* si c'est un agrégat qui ne contient pas de pointeur sur un membre non statique, pas de référence, pas de type non *POD*, pas de constructeur de copie et pas de destructeur.

Les autres termes sont définis lorsqu'ils apparaissent pour la première fois dans le document.

BIBLIOGRAPHIE

Langage C

C as a Second Language For Native Speakers of Pascal, Müldner and Steele, Addison-Wesley.

The C Programming Language, Brian W. Kernigham and Dennis M. Ritchie, Prentice Hall.

Langage C++

L'essentiel du C++, Stanley B. Lippman, Addison-Wesley.

The C++ Programming Language, Bjarne Stroustrup, Addison-Wesley.

Working Paper for Draft Proposed International Standard for Information Systems -- Programming Language C++ (http://casteyde.christian.free.fr/cpp/cours/drafts/index.html), ISO.

Bibliothèque C / appels systèmes POSIX et algorithmique

Programmation système en C sous Linux, Christophe Blaess, Eyrolles.

The Single UNIX Specification, Version 3 (http://www.unix.org/single_unix_specification/), The Open Group.

Introduction à l'algorithmique, Thomas Cormen, Charles Leiserson, et Ronald Rivest, Dunod.