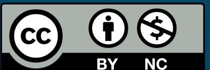
INFORMATICIE...
MEHDIDOU99



Zeste de savoir

La programmation en C++ moderne

14 mai 2019

Table des matières

I. Remerciements 12

II. Le début du voyage 14

1. Le C++, qu'est-ce que c'est ? 16
1.1. Petite mise au point . . . 16
1.1.1. Développer, un métier à part entière . . . 16
1.1.2. Votre part du travail . . . 16
1.1.3. Les mathématiques, indispensables ? . . . 17
1.1.4. L'anglais, indispensable ? . . . 17
1.2. Tour d'horizon du C++ . . . 17
1.2.1. L'histoire du C++ . . . 17
1.2.2. Pourquoi apprendre le C++ ? . . . 18
1.3. La documentation . . . 19
1.3.1. En résumé . . . 19

2. Le minimum pour commencer 20
2.1. Des outils en ligne . . . 20
2.2. Des outils plus poussés . . . 21
2.2.1. Visual Studio Community . . . 21
2.2.2. Qt Creator . . . 24
2.2.3. Pour les autres outils . . . 26
2.3. Un mot concernant Windows . . . 27
2.4. Un mot concernant GNU/Linux . . . 27
2.4.1. Un éditeur de texte . . . 28
2.4.2. Un compilateur . . . 28
2.4.3. En résumé . . . 28

3. Rencontre avec le C++ 29
3.1. Compilons notre premier programme . . . 29
3.2. Démystification du code . . . 30
3.2.1. Inclure des fonctionnalités déjà codées . . . 30
3.2.2. Le point d'entrée . . . 30
3.2.3. Voici mes instructions… . . . 31
3.3. Les commentaires . . . 32
3.3.1. La syntaxe . . . 33
3.3.2. En résumé . . . 33
Contenu masqué . . . 34

4. Une super mémoire **35**
4.1. Les littéraux . . . 35
4.1.1. Les caractères . . . 35
4.1.2. Les nombres . . . 38
4.2. Les variables . . . 40
4.2.1. Comment créer des variables en C++ ? . . . 40
4.2.2. Un peu de constance, voyons ! . . . 45
4.2.3. Manipulation de variables . . . 46
4.3. Qu'en déduisez-vous ? . . . 48
4.3.1. Avec `const` . . . 49
4.3.2. Le cas de `std::string` . . . 49
4.3.3. Les avantages . . . 50
4.4. Les entrées . . . 51
4.4.1. Gestion des erreurs . . . 51
4.4.2. En résumé . . . 53
Contenu masqué . . . 53

5. Le conditionnel conjugué en C++ **56**
5.1. Les booléens . . . 56
5.2. if — Si, et seulement si… . . . 58
5.2.1. À portée . . . 59
5.3. else — Sinon… . . . 60
5.4. else if — La combinaison des deux précédents . . . 62
5.5. Conditions imbriquées . . . 63
5.6. [T.P] Gérer les erreurs d'entrée — Partie I . . . 65
5.7. La logique booléenne . . . 66
5.7.1. AND — Tester si deux conditions sont vraies . . . 66
5.7.2. OR — Tester si au moins une condition est vraie . . . 67
5.7.3. NOT — Tester la négation . . . 68
5.8. Tester plusieurs expressions . . . 69
5.8.1. Évaluation en court-circuit . . . 71
5.9. Exercices . . . 72
5.9.1. Une pseudo-horloge . . . 72
5.9.2. Score . . . 72
5.9.3. XOR — Le OU exclusif . . . 73
5.9.4. En résumé . . . 73
Contenu masqué . . . 73

6. Des boucles qui se répètent, répètent, répètent… **80**
6.1. while — Tant que… . . . 80
6.2. Exercices . . . 83
6.2.1. Une laverie . . . 83
6.2.2. PGCD . . . 83
6.2.3. Somme de nombres de 1 à n . . . 83
6.3. do while — Répéter … tant que . . . 84
6.4. [T.P] Gérer les erreurs d'entrée — Partie II . . . 85
6.5. for — Une boucle condensée . . . 85
6.6. Boucles imbriquées . . . 87

6.7. Convention de nommage 88
6.8. Contrôler plus finement l'exécution de la boucle 88
6.8.1. break — Je m'arrête là 88
6.8.2. continue — Saute ton tour ! 90
6.8.3. En résumé 90
Contenu masqué 91

7. Au tableau ! 97
7.1. Un tableau c'est quoi ? 97
7.2. std : :vector — Mais quel dynamisme ! 97
7.2.1. Déclarer un std::vector 97
7.2.2. Manipuler les éléments d'un std::vector 99

8. Accès aux éléments 100
8.1. Exercices 108
8.1.1. Calcul de moyenne 108
8.1.2. Minimum et maximum 108
8.1.3. Séparer les pairs des impairs 109
8.1.4. Compter les occurrences d'une valeur 109
8.2. std : :array — Le tableau statique 109
8.2.1. Déclarer un std::array 109
8.2.2. Remplir un tableau 110
8.2.3. Accéder aux éléments 111
8.2.4. Connaître la taille 112
8.2.5. Vérifier si le tableau est vide 112
8.3. std : :string — Un type qui cache bien son jeu 113
8.3.1. Connaître la taille d'une chaîne 113
8.3.2. Accéder à un caractère 114
8.3.3. Premier et dernier caractère 114
8.3.4. Vérifier qu'une chaîne est vide 115
8.3.5. Ajouter ou supprimer un caractère à la fin 115
8.3.6. Supprimer tous les caractères 116
8.3.7. Boucler sur une chaîne 116
8.3.8. En résumé 117
Contenu masqué 117

9. Déployons la toute puissance des conteneurs 122
9.1. Le socle de base : les itérateurs 122
9.1.1. Déclaration d'un itérateur 122
9.1.2. Début et fin d'un conteneur 123
9.1.3. Accéder à l'élément pointé 123
9.1.4. Se déplacer dans une collection 124

10. Déplacement vers l'élément suivant 125
10.0.1. const et les itérateurs 127
10.0.2. Itérer depuis la fin 128
10.0.3. Utilisation conjointe avec les conteneurs 130
10.1. Des algorithmes à gogo ! 131
10.1.1. std::count — Compter les occurrences d'une valeur 131

10.1.2. std::find — Trouver un certain élément 132
10.1.3. std::sort — Trier une collection 134
10.1.4. std::reverse — Inverser l'ordre des éléments d'une collection 135
10.1.5. std::remove — Suppression d'éléments 136
10.1.6. std::search — Rechercher un sous-ensemble dans un ensemble 136
10.1.7. std::equal — Vérifier l'égalité de deux ensembles 137
10.1.8. Et tant d'autres 138
10.2. Personnalisation à la demande 138
10.2.1. Le prédicat, à la base de la personnalisation des algorithmes 138
10.2.2. Trier une liste dans l'ordre décroissant 138
10.2.3. Somme et produit 140
10.2.4. Prédicats pour caractères 140
10.3. Exercices 142
10.3.1. Palindrome 142
10.3.2. string_trim — Suppression des espaces 142
10.3.3. Couper une chaîne 143
10.3.4. En résumé 143
Contenu masqué 143

11. Des flux dans tous les sens **148**
11.1. Avant-propos 148
11.1.1. Prendrez-vous une extension ? 148
11.1.2. Vois sur ton chemin… 149
11.1.3. Un mot sur Windows 149
11.1.4. Pas d'interprétation, on est des brutes 150
11.2. std : :ofstream — Écrire dans un fichier 150
11.2.1. Ouvrir le fichier 151
11.2.2. Écriture 151
11.2.3. Ouvrir sans effacer 154
11.3. std : :ifstream — Lire dans un fichier 155
11.3.1. Ouvrir le fichier 155
11.3.2. Lecture 155
11.3.3. Tout lire 158
11.4. Exercice 159
11.4.1. Statistiques sur des fichiers 159
11.5. Encore plus de flux ! 159
11.5.1. Un flux c'est quoi ? 159
11.5.2. Un *buffer* dites-vous ? 160
11.5.3. Les modificateurs de flux 161
11.5.4. Même les chaînes y passent ! 162
11.5.5. En résumé 164
Contenu masqué 164

III. On passe la deuxième ! 167

12. Découpons du code — Les fonctions 169
12.1. Les éléments de base d'une fonction . . . 169
12.1.1. Une fonction, c'est quoi ? . . . 169
12.1.2. Une fonction bien connue . . . 171
12.1.3. Les composants d'une fonction . . . 171

13. Schéma d'une fonction 172
13.1. Exercices . . . 175
13.1.1. Afficher un rectangle . . . 175
13.1.2. Distributeur d'argent . . . 176
13.1.3. Parenthésage . . . 176
13.2. Quelles sont vos références ? . . . 177
13.2.1. La problématique . . . 177
13.2.2. Les références . . . 178
13.2.3. Paramètres de fonctions . . . 180
13.2.4. Valeur de retour . . . 183
13.2.5. Un mot sur la déduction de type . . . 184
13.3. Nos fonctions sont surchargées ! . . . 186
13.4. [T.P] Gérer les erreurs d'entrée — Partie III . . . 187
13.5. Dessine-moi une fonction . . . 188
13.5.1. Le problème . . . 188
13.5.2. La solution . . . 189
13.5.3. Utilisons les prototypes . . . 190
13.5.4. En résumé . . . 190
Contenu masqué . . . 191

14. Erreur, erreur, erreur… 197
14.1. L'utilisateur est un idiot . . . 197
14.2. À une condition… ou plusieurs . . . 198
14.2.1. Contrats assurés par le compilateur . . . 198

15. Le typage 199
15.0.1. Vérifier nous-mêmes . . . 199
15.1. Le développeur est un idiot . . . 199
15.1.1. Préconditions et assertions . . . 201
15.2. Les tests unitaires à notre aide . . . 202
15.2.1. Pourquoi tester ? . . . 202
15.2.2. Un test unitaire, qu'est-ce que c'est ? . . . 203
15.2.3. Écrire des tests unitaires . . . 204
15.2.4. Les tests unitaires et les postconditions . . . 205
15.3. [T.P] Gérer les erreurs d'entrée — Partie IV . . . 205
15.4. L'exception à la règle . . . 206
15.4.1. La problématique . . . 206
15.4.2. C'est quoi une exception? . . . 208
15.4.3. Lancement des exceptions dans 3, 2, 1… . . . 208
15.4.4. Attends que je t'attrape! . . . 209
15.4.5. Attrapez-les tous ! . . . 210

15.4.6. Un type pour les gouverner tous et dans le catch les lier . . . 211
15.5. [T.P] Gérer les erreurs d'entrée — Partie V . . . 211
15.5.1. En résumé . . . 212
Contenu masqué . . . 213

16. Des fonctions somme toute lambdas **214**
16.1. Une lambda, c'est quoi ? . . . 214
16.1.1. Pourquoi a-t-on besoin des lambdas ? . . . 214
16.2. Syntaxe . . . 215
16.2.1. Quelques exemples simples . . . 215
16.3. Exercices . . . 216
16.3.1. Vérifier si un nombre est négatif . . . 216
16.3.2. Tri par valeur absolue . . . 216
16.4. On va stocker ça où ? . . . 217
16.5. Paramètres génériques . . . 217
16.6. [T.P] Gérer les erreurs d'entrée — Partie VI . . . 218
16.7. [T.P] Gérer les erreurs d'entrée — Partie VII . . . 218
16.8. Capture en cours... . . . 219
16.8.1. Capture par valeur . . . 219
16.8.2. Capture par référence . . . 220
Contenu masqué . . . 221

17. Envoyez le générique ! **223**
17.1. Quel beau modèle ! . . . 223
17.1.1. La bibliothèque standard : une fourmilière de modèles . . . 225
17.1.2. Un point de vocabulaire . . . 225
17.2. [T.P] Gérer les erreurs d'entrée - Partie VIII . . . 225
17.3. Instanciation explicite . . . 226
17.4. Ce type là, c'est un cas ! . . . 227
17.4.1. En résumé . . . 228
Contenu masqué . . . 228

18. De nouvelles structures de données **230**
18.1. struct — Un agrégat de données . . . 230
18.1.1. Stockage d'informations personnelles[1] . . . 230
18.1.2. Analyse de l'exercice . . . 231
18.1.3. Introduction aux structures . . . 231
18.1.4. Exercice . . . 232
18.2. std : :tuple — Une collection hétérogène . . . 233
18.2.1. Déclaration . . . 233
18.2.2. Accès aux éléments . . . 233
18.2.3. `std::tuple` et fonctions . . . 234
18.2.4. Équations horaires . . . 234
18.3. std : :unordered_map — Une table associative . . . 235
18.3.1. Un dictionnaire de langue Zestienne . . . 236
18.3.2. Toujours plus de clés . . . 237
18.3.3. Cette clé est unique ! . . . 237
18.3.4. Cherchez un élément . . . 238
18.3.5. Un exemple concret . . . 238

18.4. std : :unordered_set — Représenter un ensemble 238
18.4.1. Inscription sur Zeste de Savoir 239
18.5. Un peu d'ordre, voyons ! 239
18.6. Une longue énumération 240
18.6.1. En résumé 241
Contenu masqué 241

19. Reprendriez-vous un peu de sucre syntaxique ? 243
19.1. Ce type est trop long ! 243
19.2. Décomposons tout ça 243
19.2.1. En pleine décomposition 244
19.2.2. Et avant, on faisait comment ? 244
19.3. La surcharge d'opérateurs 245
19.3.1. Le problème... 245
19.3.2. ...et la solution 246
19.3.3. Cas concret — Les fractions 246

20. Les opérateurs arithmétiques 247
20.0.1. Et la bibliothèque standard ? 249
20.0.2. Exercice — Calculs avec des durées 250
20.0.3. Sur le bon usage de la surcharge 251
20.0.4. En résumé 251
Contenu masqué 252

21. [T.P] Un gestionnaire de discographie 253
21.1. L'énoncé 253
21.1.1. Ajout de morceaux 253
21.1.2. Affichage de la discographie 254
21.1.3. Enregistrement et chargement d'une discographie 255
21.1.4. Quitter le programme 255
21.1.5. Gestion des erreurs 255
21.1.6. Dernières remarques 256
21.2. Guide de résolution 256
21.2.1. Analyse des besoins 256
21.2.2. Attaquons le code ! 257
21.3. Corrigé complet 260
21.4. Conclusion et pistes d'améliorations 261
21.4.1. En résumé 261
Contenu masqué 262

22. Découpons du code — Les fichiers 265
22.1. Le principe 265
22.1.1. Le fichier d'en-tête 265
22.1.2. Le fichier source 271
22.1.3. Créons un lien 271
22.2. La sécurité, c'est important 272
22.3. Découpons le TP ! 273
22.4. Les avantages du découpage en fichiers 274
22.4.1. Une structure de projet plus visible 274

22.4.2. Une meilleure abstraction . . . 275
22.4.3. Une meilleure modularité . . . 275
22.5. Le cas des templates . . . 275
22.5.1. En résumé . . . 277
Contenu masqué . . . 277

IV. Interlude - Être un développeur 280

23. Avant-propos 282
23.1. Ce que nous avons appris . . . 282
23.2. Différents paradigmes . . . 283
23.2.1. Le paradigme impératif . . . 283
23.2.2. Le paradigme fonctionnel . . . 283
23.2.3. Le paradigme générique . . . 284
23.2.4. La programmation par contrat . . . 284
23.3. Brancher le cerveau avant tout . . . 285
23.4. Savoir se débrouiller . . . 286
23.4.1. En résumé . . . 286

24. Mais où est la doc ? 288
24.1. Lire une page de doc . . . 288
24.1.1. À l'arrivée . . . 288
24.1.2. `vector` — Retour sur le plus célèbre des conteneurs . . . 289

25. Présentation 290
25.0.1. Les algorithmes . . . 292
25.0.2. Les chaînes de caractère au grand complet . . . 293
25.1. Exercices . . . 294
25.1.1. Remplacer une chaîne de caractère par une autre . . . 294
25.1.2. Norme d'un vecteur . . . 294
25.1.3. Nombres complexes . . . 295
25.1.4. Transformations . . . 295
25.2. Documenter son code avec Doxygen . . . 295
25.2.1. Installation des outils . . . 295
25.2.2. Écrire la documentation . . . 299
25.3. Quelques bonnes pratiques . . . 302
25.3.1. Ce qu'il faut documenter . . . 302
25.3.2. Les fichiers d'en-tête ou source ? . . . 303
25.3.3. Prenons un exemple . . . 303
25.3.4. Commentaire vs documentation . . . 303
25.3.5. En résumé . . . 305
Contenu masqué . . . 305

26. Compilation en cours… 308
26.1. Le préprocesseur . . . 308
26.1.1. Inclure des fichiers . . . 308
26.1.2. Conditions . . . 308
26.1.3. *Debug* ou *release* ? . . . 309

26.2. La compilation . . . 310
26.2.1. Les templates . . . 311
26.2.2. `constexpr` . . . 311
26.2.3. La compilation à proprement parler . . . 313
26.2.4. Une étape intermédiaire cachée . . . 314
26.2.5. Influer sur la compilation . . . 314
26.3. Le linker . . . 317
26.3.1. Une question de symboles . . . 318
26.4. Schéma récapitulatif . . . 319
26.4.1. En résumé . . . 320

27. Chasse aux bugs ! 322
27.1. Le principe . . . 322
27.2. Visual Studio . . . 323
27.2.1. Interface . . . 323
27.2.2. Pas-à-pas . . . 325
27.2.3. Point d'arrêt conditionnel . . . 326
27.3. Qt Creator . . . 327
27.3.1. Interface . . . 328
27.3.2. Pas-à-pas . . . 329
27.3.3. Point d'arrêt conditionnel . . . 330
27.3.4. Aller plus loin . . . 331
27.4. En ligne de commande avec gdb . . . 332
27.4.1. Poser un point d'arrêt . . . 332
27.4.2. Supprimer des points d'arrêt . . . 333
27.4.3. Désactiver des points d'arrêt . . . 333
27.4.4. Afficher l'état d'une variable . . . 333
27.4.5. Pas-à-pas . . . 333
27.4.6. Conditions . . . 334
27.4.7. Aller plus loin . . . 334
27.4.8. En résumé . . . 334

28. Une foule de bibliothèques 335
28.1. Quelles bibliothèques choisir ? . . . 335
28.1.1. Boost . . . 335
28.1.2. SFML . . . 336
28.2. Généralités . . . 336
28.2.1. Statique ou dynamique ? . . . 336
28.2.2. *Debug* ou *release* ? . . . 337
28.3. Installer Boost . . . 337
28.3.1. GNU/Linux . . . 337
28.3.2. Windows . . . 337
28.3.3. Tester l'installation . . . 340
28.4. Installer SFML . . . 341
28.4.1. GNU/Linux . . . 341
28.4.2. Windows . . . 342
28.4.3. Tester l'installation . . . 345
28.4.4. En résumé . . . 345

29. Améliorer ses projets **346**
29.1. git — Sauvegarder et versionner son code . . . 346
29.1.1. La problématique . . . 346
29.1.2. Installation . . . 346
29.1.3. Initialiser git . . . 347
29.1.4. Créer un dépôt . . . 347
29.1.5. Connaître l'état de ses fichiers . . . 348
29.1.6. Ajouter un fichier . . . 348
29.1.7. Valider les modifications . . . 348
29.1.8. Voir l'historique . . . 349
29.1.9. Bloquer certains fichiers . . . 349
29.1.10. Aller plus loin . . . 350
29.1.11. Visual Studio . . . 350
29.2. GitHub — Partager son code . . . 352
29.2.1. La problématique . . . 352
29.2.2. Création d'un projet . . . 353
29.2.3. Récupérer localement les modifications . . . 355
29.2.4. Pousser nos modifications . . . 355
29.2.5. Aller plus loin . . . 356
29.3. CMake — Automatiser la compilation de nos programmes . . . 356
29.3.1. La problématique . . . 356
29.3.2. Un exemple simple . . . 357
29.3.3. Lier des bibliothèques externes . . . 358
29.3.4. Définir des variables au lancement . . . 362
29.3.5. Aller plus loin . . . 362
29.4. Aller plus loin . . . 362
29.4.1. TravisCI — De l'intégration continue . . . 363
29.4.2. CppCheck — Vérification du code . . . 363
29.4.3. StackOverflow — La réponses à quasiment toutes les questions . . . 363
29.4.4. En résumé . . . 363
Contenu masqué . . . 364

V. [EdC] Zesty Creatures, un Pokémon-like en console ! **365**

VI. La Programmation Orientée Objet **366**

Bienvenue dans ce cours sur l'apprentissage de la programmation avec le langage C++. Vous débutez complètement et n'avez qu'une vague idée de ce que peut bien être la programmation ? On vous a dit d'en apprendre plus sur le C++ ? Alors **ce tutoriel est fait pour vous**.

Pour rendre ce cours **pédagogique**, nous alternerons entre théorie, explications et détails techniques d'un côté et exercices, travaux pratiques et questions-réponses de l'autre. L'accent sera mis sur **la rigueur**, **la qualité** et les **bonnes pratiques**. Le rythme sera **progressif**, chaque concept étant introduit en temps voulu et en lien avec ce que nous aurons déjà vu.

Prérequis

Prérequis
Lire cette introduction à la programmation afin d'avoir des bases pour aborder l'apprentissage du C++.
Objectifs
Vous former à la programmation en vous donnant des bases solides sur lesquelles continuer.
Vous former au C++ moderne.

Première partie

Remerciements

Car ce cours n'aurait jamais pu voir le jour sans l'aide et le soutien de plusieurs personnes.

— @Akulen pour sa participation à la rédaction dans ses débuts.
— Un merci particulier à @gbdivers pour son cours Débuter en C++ moderne qui aura été une source d'inspiration pour l'écriture du présent cours et à qui j'ai piqué plusieurs passages très bien expliqués.
— Tous les membres qui ont participé à la bêta et qui ont grandement aidé à améliorer la qualité de ce tutoriel. Je cite en particulier @lmghs pour tous les commentaires et idées qu'il a soulevés, ainsi que @gbdivers.
— Toute l'équipe de Zeste de Savoir ; en particulier, un grand merci à @Taurre qui s'est chargé de la validation.
— Et l'ensemble des lecteurs, pour avoir choisi ce cours.

! À venir

Ce cours est toujours en rédaction. D'autres chapitres et sections suivront au fur et à mesure.

Deuxième partie

Le début du voyage

Vous y voilà. Grâce au tutoriel sur les bases de la programmation , vous en savez un peu plus sur le merveilleux monde de la programmation. Maintenant, vous avez hâte d'appliquer tout ça concrètement.

Le but de cette première partie est de vous introduire aux concepts de bases de la programmation. Ce que vous y apprendrez vous permettra de faire des programmes assez utiles. Bien entendu, votre apprentissage ne s'arrêtera pas de si tôt.

1. Le C++, qu'est-ce que c'est ?

Voilà, vous êtes décidés, vous voulez apprendre à programmer à l'aide du langage C++. Mais vous devez toujours avoir **quelques questions** qui trottent dans votre esprit.

— Concrètement, C++, c'est quoi ?
— Et puis pourquoi commencer par C++ ?
— Que vais-je savoir faire en sortant de ce tutoriel ?
— Il y a-t-il des conditions, des prérequis, pour pouvoir suivre ce cours ?

1.1. Petite mise au point

J'ai indiqué en introduction que le seul prérequis était de lire l'introduction à la programmation et c'est vrai. Mais si peu de connaissances sont nécessaires pour comprendre tout ce qui sera dit, cela ne veut pas dire pour autant que ce sera du tout cuit, prémâché, directement dans la bouche. C'est pour cela que je veux prendre quelques minutes pour clarifier ceci.

1.1.1. Développer, un métier à part entière

Eh oui ! Créer des logiciels, **c'est un métier**. D'ailleurs, il y a mêmes des écoles et des études spécialisées pour ça. C'est que développer, ce n'est pas simplement écrire du code. Il y a aussi des phases de réflexion, de conceptions, d'écriture, de validation, de tests, de réécriture d'anciennes portions, etc.

Par ailleurs, même si cela n'est pas directement lié à la programmation, peut également s'ajouter à cela : la gestion de base de données, l'usage du réseau, le management, la gestion de projet, etc. En bref, **être développeur c'est beaucoup de compétences différentes dans des domaines variés**.

La cruelle vérité qui se cache derrière tout ça, c'est que **ce cours ne fera pas de vous des experts, ni des développeurs professionnels**. Par contre, une fois fini, vous aurez des bases solides pour continuer votre apprentissage. La route du savoir est infinie.

1.1.2. Votre part du travail

Le cours est écrit de façon à être le plus clair possible, sans vous noyer sous un flot de détails et d'explications, mais il arrivera parfois que vous ne compreniez pas un morceau de code ou une explication. C'est tout à fait normal, ça fait partie de l'apprentissage. Reprenez le cours à tête reposée, aidez-vous de schémas ou de dessins, demandez de l'aide sur les forums , et vous ne resterez jamais bloqués longtemps.

Par contre, il faut être prêt à **fournir des efforts** de votre côté. Cela signifie ne pas se ruer sur les forums au bout de quelques minutes, sans même avoir essayé de trouver une solution.

1.1.3. Les mathématiques, indispensables ?

Une des grandes appréhensions, qui revient régulièrement dans la bouche des débutants, est de savoir si les mathématiques sont un prérequis à l'apprentissage de la programmation. La réponse est **non**. Il est tout à fait possible d'**apprendre à programmer tout en ayant un faible niveau en mathématique**. Ce cours ne demande aucune connaissance en mathématiques plus poussées que les opérations de base et, quelques fois, l'utilisation de sinus et cosinus, en guise d'exemples.

Bien sûr, certains aspects de la programmation, comme la sécurité, la cryptographie ou les applications scientifiques vont demander un bagage mathématique solide. Mais cela ne rentre pas dans le cadre de ce cours.

1.1.4. L'anglais, indispensable ?

À strictement parler, dans le cadre de ce cours, **pas besoin de savoir parler anglais**. Même si nous examinerons régulièrement de la documentation écrite dans la langue de Shakespeare, je serai là pour donner des explications et éclaircissements. Et quand bien même il y aurait quelque chose sur lequel vous butez, vous pouvez vous servir d'un traducteur automatique.

D'un point de vue plus général, si vous souhaitez continuer dans l'informatique et la programmation, il sera très difficile d'échapper à l'anglais. Beaucoup de cours, de documents, de forums sont en anglais, ou en font un usage massif. L'anglais est tout simplement **la langue de l'informatique**. Après, sachez que l'anglais informatique est **simple à comprendre**, car souvent écris par des gens dont l'anglais n'est pas la langue maternelle. Ainsi, inutile d'être bilingue, un bon dictionnaire ou un traducteur vous aideront.

1.2. Tour d'horizon du C++

1.2.1. L'histoire du C++

Faisons un plongeon dans l'histoire et revenons dans les années 1970. À cette époque, Dennis Ritchie, programmeur aux laboratoires AT&T aux États-Unis, invente le langage C, conçu pour programmer le système d'exploitation UNIX. Ce langage devint très populaire à tel point qu'il est encore beaucoup utilisé aujourd'hui, dans l'écriture de Linux par exemple. Puis un peu plus tard, au début des années 1980, Bjarne Stroustrup, lui aussi développeur aux laboratoires AT&T, décida de prendre le langage C comme base et de lui ajouter des fonctionnalités issues d'un autre langage appelé Simula. Ce langage devint alors le **C with classes**.

Finalement, en 1983, son créateur, estimant que le nom de son langage était trop réducteur aux vues de tous les ajouts faits par rapport au C, décida de le renommer **C++**. Mais l'histoire ne s'arrête pas là. Au contraire, le C++ continue d'évoluer à tel point qu'on décide au début des années 1990 de **le normaliser**, c'est-à-dire d'en établir les règles officielles. Ce travail de longue

haleine s'acheva en 1998 ; cette version est ainsi souvent nommée C++98. Ensuite, en 2003, des corrections ont été apportées et l'on obtint C++03.

Puis de nouveau un chantier titanesque est mis en place pour améliorer encore plus le C++, ce qui aboutit 8 ans plus tard, en 2011, à la sortie de C++11, jugée par beaucoup de développeurs comme étant la renaissance du C++. Ensuite, de nouvelles corrections et quelques ajustements ont été apportés pour donner C++14. Enfin, à l'heure actuelle, la norme C++17, nouvelle version majeure et apportant tout un lot de choses intéressantes, est sortie en fin d'année et C++20 est déjà en chantier.

> i
> Beaucoup de programmeurs utilisent le terme « **C++ historique** » pour désigner les normes C++98 et C++03 et le terme « **C++ moderne** » pour parler de C++11 et au-delà.

1.2.2. Pourquoi apprendre le C++ ?

- Sa **popularité** : le C++ est un langage qui est utilisé dans de nombreux projets important (citons Libre Office, 7-zip ou encore KDE). Il est au programme de beaucoup de formations informatiques. Il possède une communauté très importante, beaucoup de documentation et d'aide, surtout sur l'internet anglophone.
- Sa **rapidité** : C++ offre un grand contrôle sur la rapidité des programmes. C'est cette caractéristique qui fait de lui un des langages de choix pour les programmes scientifiques, par exemple.
- Sa **facilité d'apprentissage** : depuis sa version de 2011, C++ est beaucoup plus facile à apprendre que par le passé. Et ça tombe bien, c'est sur cette version et les suivantes que va se baser ce cours.
- Son **ancienneté** : C++ est un langage ancien d'un point de vue informatique (30 ans, c'est énorme), ce qui donne une certaine garantie de maturité, de stabilité et de pérennité (il ne disparaîtra pas dans quelques années).
- Son **évolution** : C++11 est un véritable renouveau de C++, qui le rend plus facile à utiliser et plus puissant dans les fonctionnalités qu'il offre aux développeurs. C++14 et C++17 améliorent encore la chose.
- Il est **multi-paradigme** : il n'impose pas une façon unique de concevoir et découper ses programmes mais laisse le développeur libre de ses choix, contrairement à d'autres langages comme Java ou Haskell.

Bien entendu, tout n'est pas parfait et C++ a aussi ses défauts.

- Son **héritage du C** : C++ est un descendant du langage C, inventé dans les années 1970. Certains choix de conception, adaptés pour l'époque, sont plus problématiques aujourd'hui, et C++ les traine avec lui.
- Sa **complexité** : il ne faut pas se le cacher, avoir une certaine maîtrise du C++ est très long et demandera des années d'expérience, notamment parce que certaines des fonctionnalités les plus puissantes du C++ requièrent de bien connaître les bases.
- Sa **bibliothèque standard** : bien qu'elle permette de faire beaucoup de choses (et d'ailleurs, nous n'aurons même pas le temps d'en faire un tour complet dans ce cours), elle n'offre pas de mécanisme natif pour manipuler des bases de données, faire des

programmes en fenêtres, jouer avec le réseau, etc. Par rapport à d'autres langages comme Python , C++ peut paraître plus « limité ».

1.3. La documentation

En programmation, il y a un réflexe à adopter le plus rapidement possible : si on ne sait pas comment utiliser un outil, il faut aller consulter **la documentation** de l'outil concerné, et ce avant de demander de l'aide sur un forum par exemple. Voici un lien vers une excellente documentation C++. Elle est en anglais, mais pas de soucis, je suis là avec vous. Je vous donnerai des liens, vous expliquerai comment comprendre et exploiter les informations fournies pour que, par la suite, vous puissiez le faire vous-mêmes.

Il y a aussi un autre outil, très utile pour rechercher une information et que vous connaissez déjà, **les moteurs de recherches** (Google, Bing, DuckDuckGo, Qwant, etc). Sachez aussi que **les forums** sont une mine d'informations. Vous pouvez par exemple utiliser ceux de Zeste de Savoir , en n'oubliant bien évidemment pas d'utiliser le tag `[c++]` lors de la création d'un sujet.

Enfin, sachez qu'il existe une référence ultime appelée **la norme**, produit par un organisme de validation international appelé l'ISO, qui explique tous les détails et les règles du C++ mais qui est un document complexe et très largement hors de portée pour vous.

La dernière version de ce document est sortie en 2017 et explique les règles de fonctionnement du C++ dans sa version de 2017. Je le mentionne simplement pour que vous soyez au courant de son existence, sans être surpris si, lors de vos recherches sur Internet, des réponses mentionnent ou citent la norme.

1.3.1. En résumé

- La programmation est une activité parfois complexe, mais très enrichissante et accessible.
- Le cours se veut le plus accessible possible, mais vous devez jouer le jeu et faire des efforts de votre côté.
- Les mathématiques et l'anglais ne sont pas requis pour suivre ce cours, bien qu'un niveau acceptable en anglais soit extrêmement utile par la suite.
- Le C++ a une longue histoire, mais le C++ moderne a véritablement vu le jour en 2011, avec des améliorations et corrections apportées en 2014 et 2017.
- Le C++ possède des forces qui en font un des langages les plus utilisés sur la planète.
- En contrepartie, le C++ est un langage assez complexe qui demande de nombreuses années avant d'être « maitrisé ».
- La documentation nous aide à mieux comprendre le langage, son fonctionnement, etc. Elle est en anglais, mais illustrée par des exemples.

2. Le minimum pour commencer

Maintenant que nous sommes bien au fait de ce que nous allons apprendre, il est temps de se pencher sur quelques outils que nous utiliserons. Nous verrons tant les **outils en ligne** pour tester rapidement un morceau de code que les **outils de développement complets**, à installer sur votre ordinateur.

2.1. Des outils en ligne

Nous l'avons vu dans l'introduction à la programmation, il existe des langages compilés et C++ en fait partie. Nous allons donc avoir besoin d'un **compilateur**. Mais des compilateurs C++, il en existe beaucoup. Et vous, lecteurs, n'avez pas tous les mêmes systèmes d'exploitation (Windows, Mac OS, GNU/Linux, BSD, etc) ni les mêmes versions (Windows XP, 7, 8, 10, distributions GNU/Linux, etc).

Aujourd'hui, grâce à Internet, nous pouvons accéder à de nombreuses ressources dont des compilateurs C++ dans leur dernière version. L'un deux s'appelle **Wandbox**. Ce site fournit des compilateurs en ligne pour de nombreux langages, dont deux pour C++ qui sont parmi les plus connus : Clang et GCC.

Je suis conscient que cela demande un accès constant à Internet. La contrepartie est que l'on peut ainsi pratiquer partout (école, travail, ordinateur de quelqu'un d'autre) puisque aucune installation n'est requise. Nous verrons plus loin dans le cours quelques outils plus avancés que vous pourrez installer sur votre ordinateur.

2.1.0.1. Survol de Wandbox

— En haut à gauche, dans un encart vert, c'est le **langage de programmation utilisé**. Si un autre nom que C++ est écrit, pas de soucis, il suffit de cliquer sur la liste déroulante en dessous, puis sur « C++ » et choisir « **GCC HEAD** » ou « **Clang HEAD** » (qui sont les toutes dernières versions de ces compilateurs).

— Un peu plus bas, la liste déroulante commençant par C++ permet de **choisir la version du langage**. Dans ce tutoriel, nous utiliserons la version « **C++17** », qui correspond à la version 2017 du langage. Que vous choisissiez « C++17 » ou « C++17 (GNU) », cela n'a aucune importance à notre niveau.

— La grande zone de texte blanche est **là où nous écrirons le code**.

— Enfin, le bouton Run (or Ctrl + Enter) servira à lancer le compilateur et lui faire compiler le code écrit dans l'encadré précédent.

2.2. Des outils plus poussés

Les outils en ligne sont très bien pour des petits codes rapides ou pour faire quelques tests, mais ils sont quand même limités. Sachez qu'il existe cependant des outils très complets tant pour Windows que pour GNU/Linux. Comme mentionné dans le cours d'introduction à la programmation , on peut utiliser un ensemble d'outils et de logiciels différents ou bien utiliser un IDE.

2.2.1. Visual Studio Community

L'IDE par excellence de Microsoft, qui permet de programmer non seulement en C++, mais aussi en Python, en JavaScript, en C#, etc. Configurable, extensible, léger par défaut dans sa version 2017, il est en plus **100% gratuit dans le cadre de l'apprentissage**, de la recherche ou des projets *open source*. Vous n'aurez donc pas besoin de débourser quoi que ce soit pour l'utiliser. Inconvénient ? Il n'est **disponible que pour Windows** (dans sa version 7 au minimum).

Pour le télécharger et l'installer, je vous invite à suivre la documentation fournie par Microsoft . L'installation demande une connexion Internet.

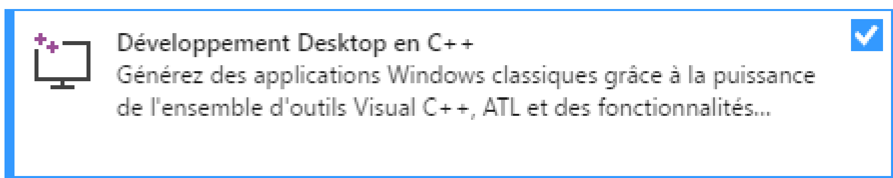


FIGURE 2.1. – Vérifiez que la case C++ est bien cochée.

Une fois que Visual Studio est installé pour la première fois, il va vous demander quelques informations supplémentaires, notamment de vous connecter. C'est gratuit et nécessaire pour activer le logiciel. Ensuite, choisissez la couleur de votre IDE et vous arriverez sur l'écran d'accueil.

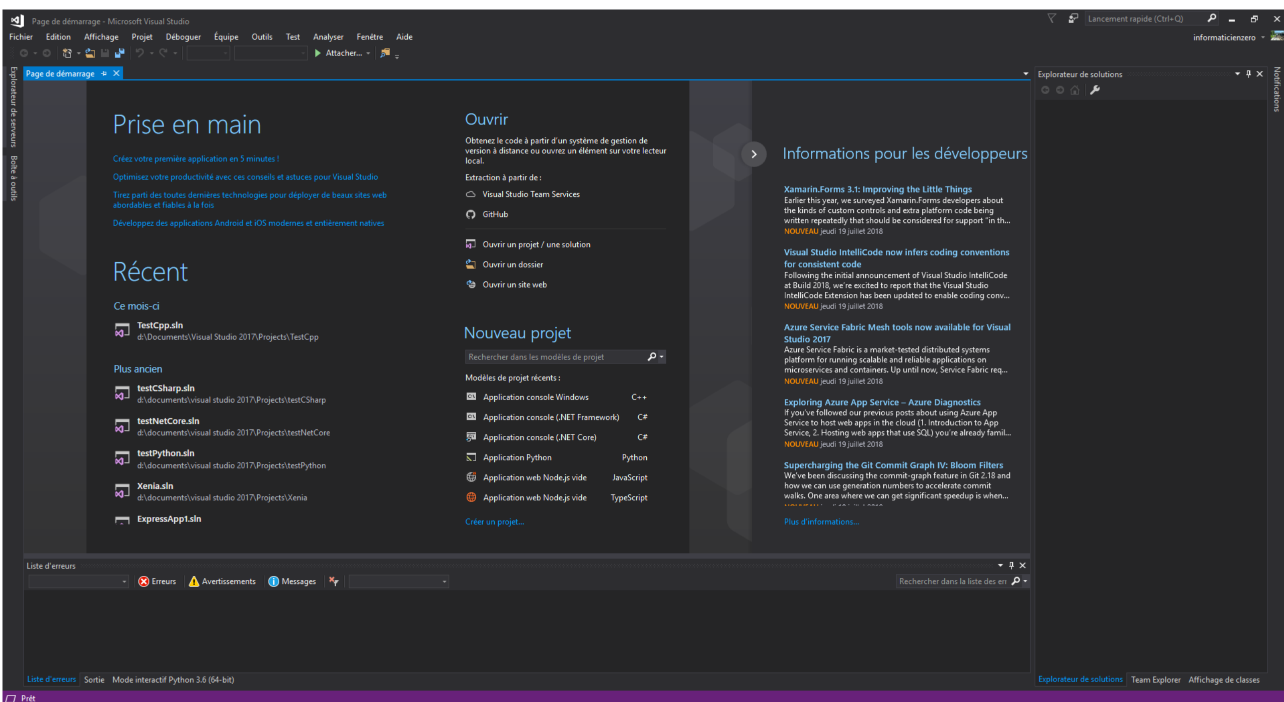


FIGURE 2.2. – L'écran d'accueil de Visual Studio, avec le thème sombre.

Pour créer un projet, cliquez sur `Fichier -> Nouveau -> Projet...`, puis sur `Modèle(s) -> Visual C++ -> Général -> Projet vide`. Là, donnez un nom qui vous plaît au projet puis validez.

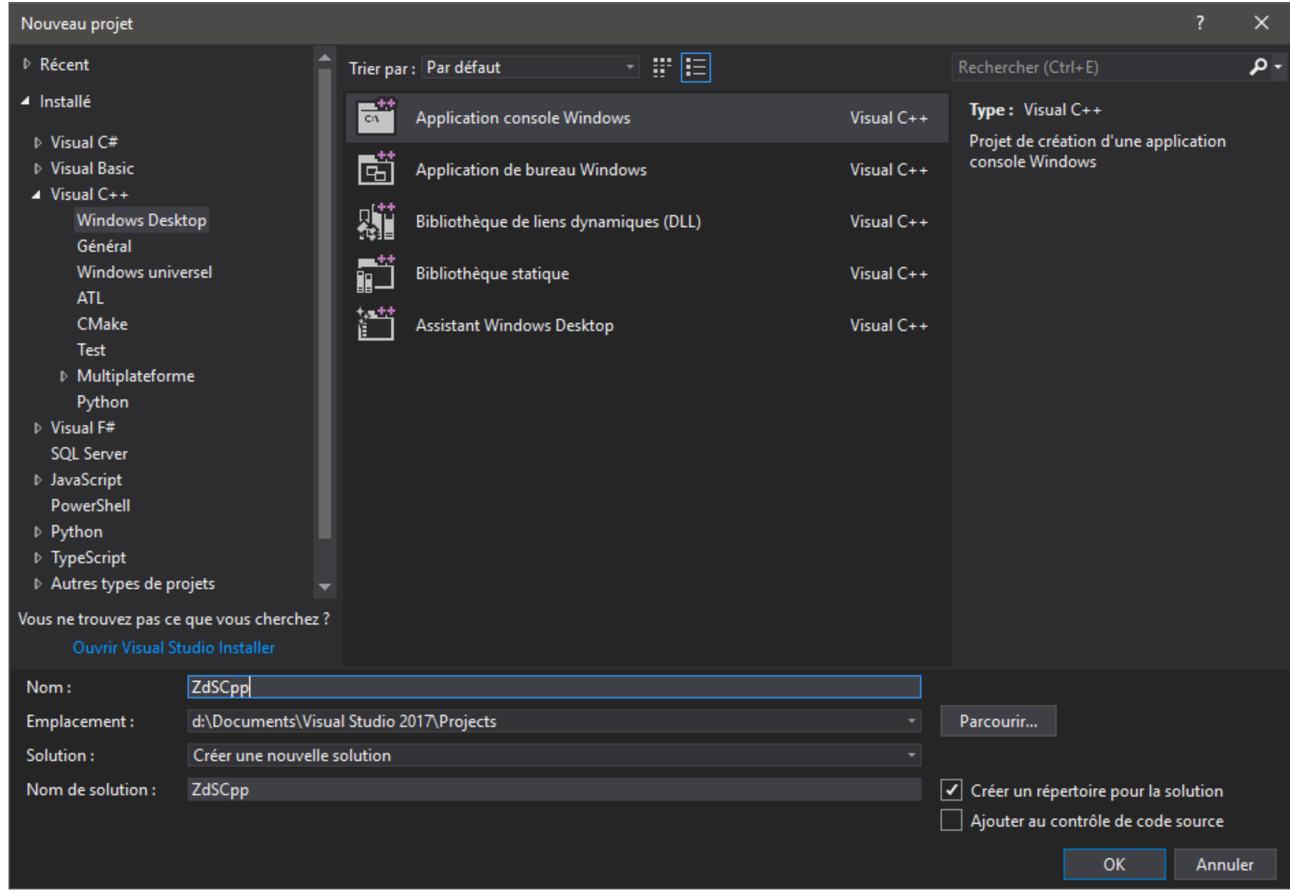


FIGURE 2.3. – Voici la fenêtre de création d'un nouveau projet.

Une fois que votre projet est créé, faites un clic-droit sur Fichiers sources, à droite de l'écran, puis Ajouter -> Nouvel élément…. Cliquez sur Fichier C + (.cpp), nommez le fichier **main.cpp** et validez. Ensuite, supprimez tous les autres fichiers **.cpp** et tous les fichiers **.hpp** du dossier **Fichiers d'en-tête**.

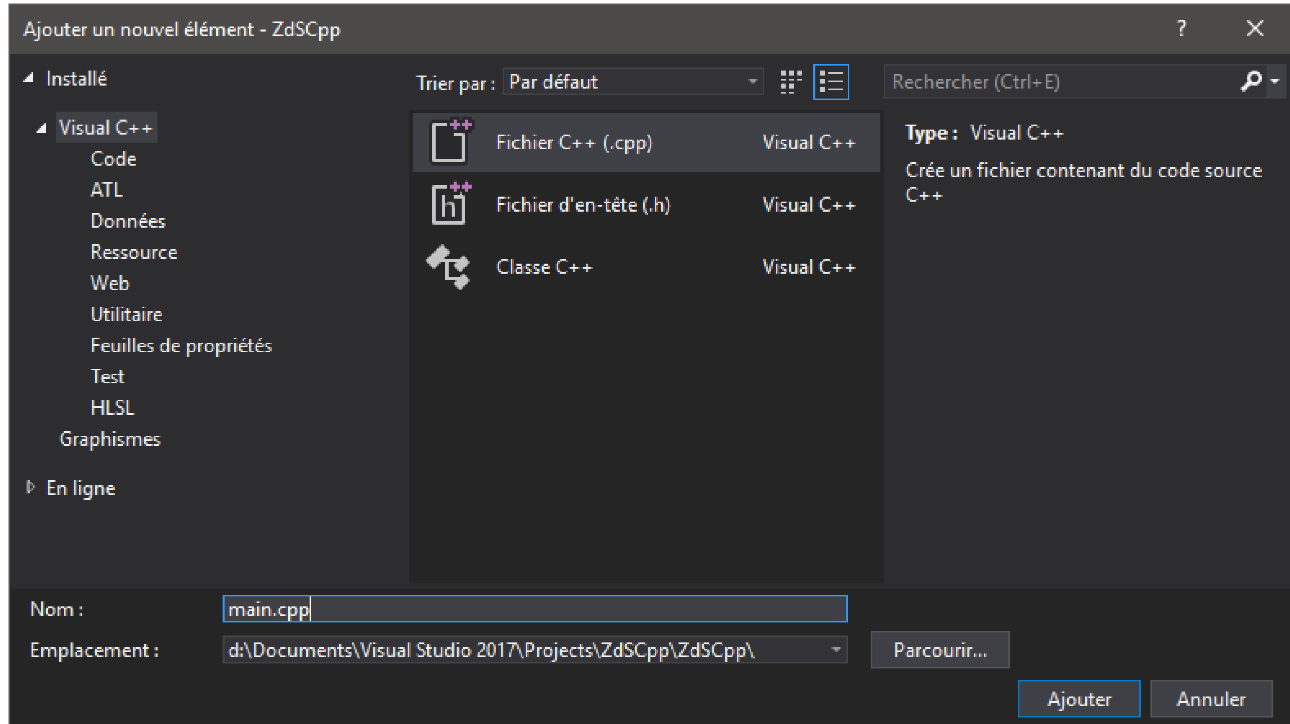


FIGURE 2.4. – Ajout d'un fichier C++.

Il ne reste plus qu'à régler quelques paramètres du projet pour utiliser la dernière norme de C++ et nous serons bons. Pour cela, à droite, faîtes un clic-droit sur le nom de votre projet, puis descendez pour cliquer sur Propriétés. Tout en haut, mettez l'option Configuration à la valeur Toutes les configurations et l'option Plateforme à la valeur Toutes les plateformes.

FIGURE 2.5. – Ce que vous devez avoir comme valeurs en haut de la fenêtre.

Dans Propriétés de configuration, déroulez C/C puis Langage. À la ligne **Norme du langage C++**, sélectionnez **Norme ISO C++17 (/std :c++17)** et validez.

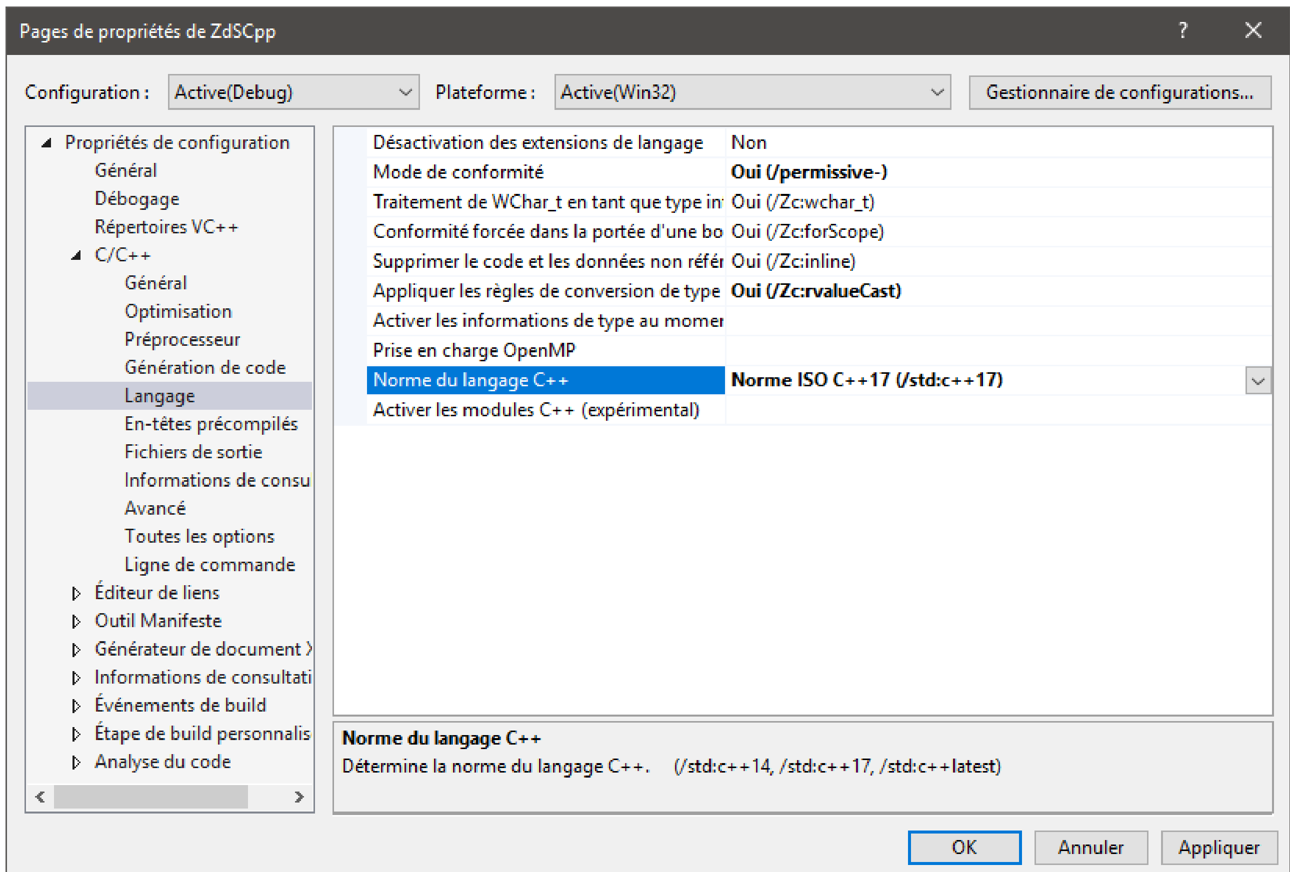


FIGURE 2.6. – Voici la fenêtre de sélection de la norme C++.

Un deuxième paramètre à modifier se trouve dans la rubrique C/C++ -> En-têtes précompilés. À la rubrique **En-têtes précompilés**, attribuez la valeur **Sans utiliser les en-têtes précompilés**.

Le dernier est toujours dans Propriétés -> Propriétés de configuration, mais sous la rubrique Éditeur de liens -> Système. Là, attribuez à la rubrique **Sous-système** la valeur **Console (/SUBSYSTEM :CONSOLE)**, puis validez.

2.2.2. Qt Creator

Un autre grand IDE pour programmer en C++ se nomme Qt Creator, disponible sous Windows, MacOS et GNU/Linux. Il est fourni par une compagnie du nom de Qt, qui édite également une bibliothèque C++ très populaire et répandue, que vous aurez peut-être l'occasion de croiser ou d'apprendre une fois ce cours terminé.

Téléchargez la version Open Source, puis installez-la. Sélectionnez la **dernière version de Qt disponible** et, dans *Tools*, prenez **la version la plus à jour de MinGW**. L'installation peut être plus ou moins longue, en fonction de la qualité de votre connexion Internet notamment. Si vous avez des problèmes, vous pouvez suivre le tutoriel de @gbdivers, qui détaille chaque étape de l'installation.

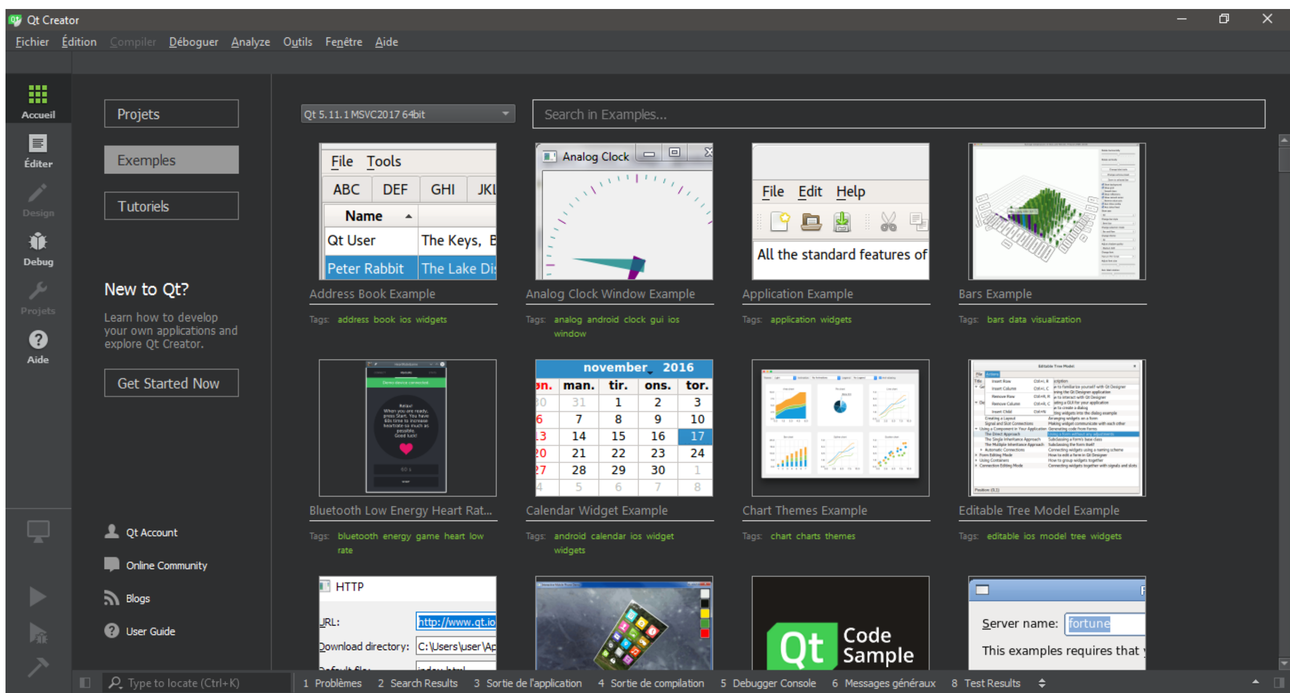


Figure 2.7. – Voici l'écran d'accueil, au lancement de Qt Creator.

Pour créer un nouveau projet, cliquez sur Fichier -> Nouveau fichier ou projet…, puis dans Projets -> Non-Qt Project -> P + Application.

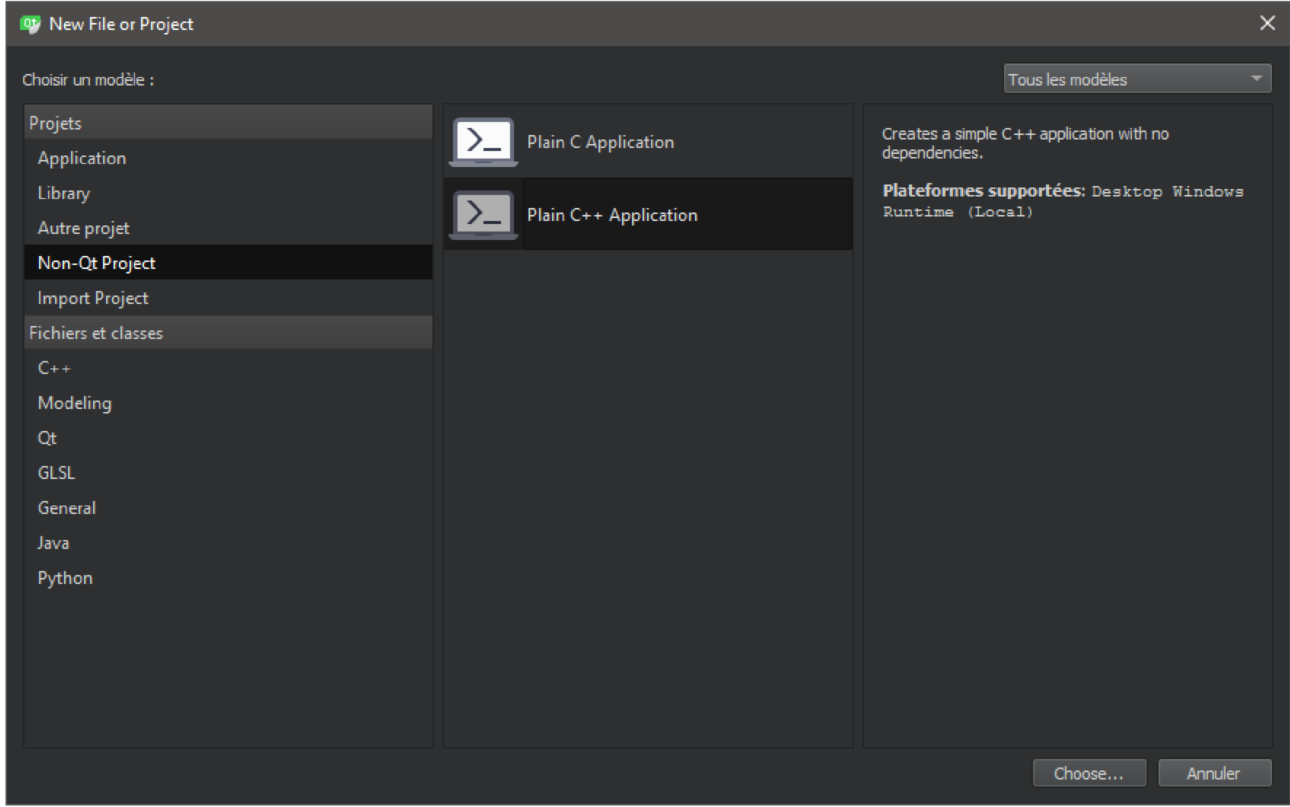


Figure 2.8. – Écran de choix du projet.

Cliquez ensuite sur Choose. L'écran d'après vous demande le nom du projet et où il sera localisé.

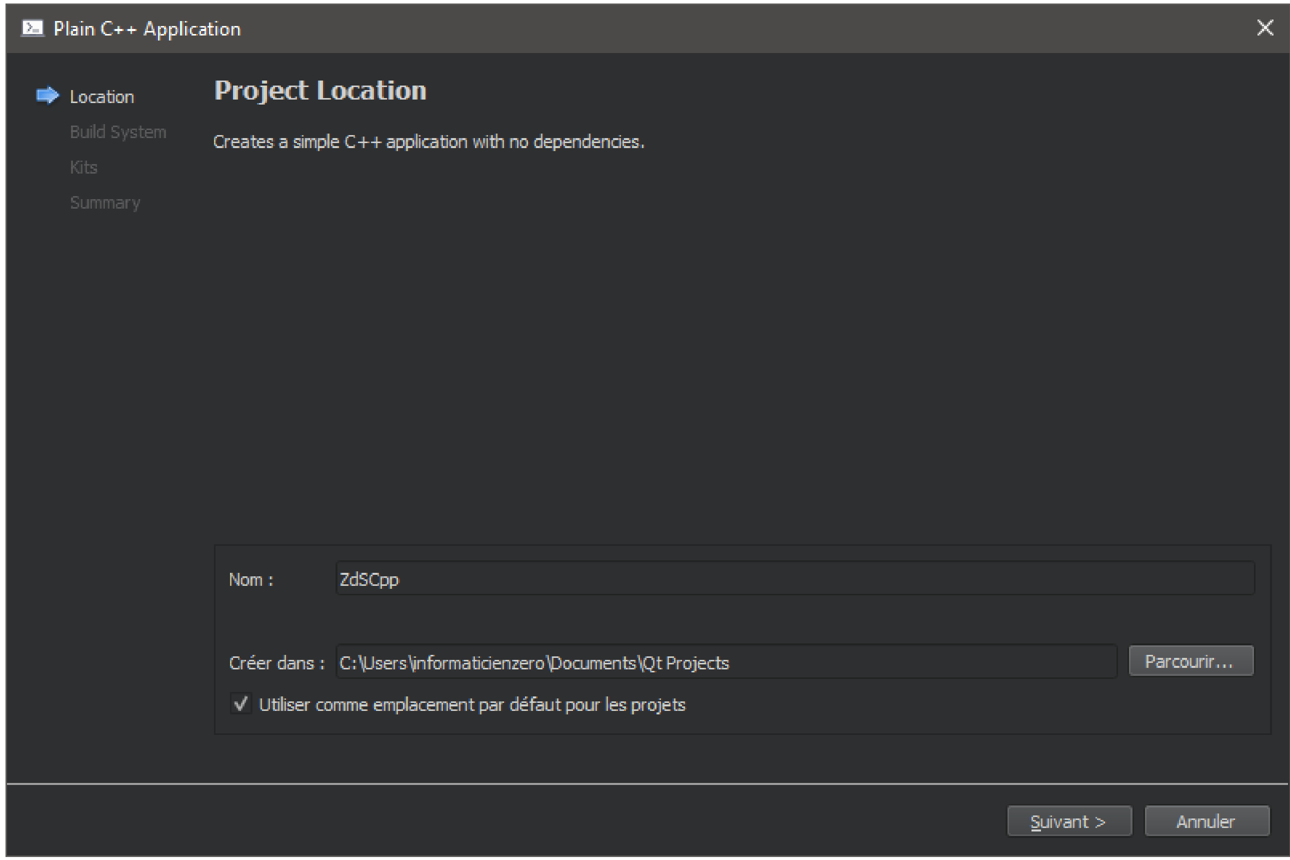


FIGURE 2.9. – Choix du nom et de la localisation du projet.

Sur les écrans d'après, laissez les choix par défaut et cliquez à la fin sur Terminer. Ouvrez enfin le fichier se terminant par **.pro** et ajouter la première ligne pour une version de Qt inférieur à 5.12, la deuxième à partir de Qt 5.12.

```
CONFIG += c++1z
```

```
CONFIG += c++17
```

2.2.3. Pour les autres outils

Si vous souhaitez utiliser un autre outil que l'un de ces deux, vous pouvez, bien-sûr. Par contre, vous devrez vous débrouiller pour le configurer, car on ne peut malheureusement pas expliquer la procédure pour tous les outils existants.

Pour les utilisateurs de XCode, vous avez de la chance, un lecteur a fait un post décrivant la procédure à suivre .

2.3. Un mot concernant Windows

Un petit paragraphe pour ceux d'entre vous qui tournent sous Windows, et ce peu importe que vous utilisiez Visual Studio ou Qt Creator. Par défaut, **la console Windows ne sait pas bien gérer les accents de notre belle langue française**. Heureusement, c'est très simple à changer.

Dans le menu Démarrer, tapez Cmd et ouvrez l'application. Ensuite, faites un clic droit sur la barre des tâches puis cliquez sur Propriétés, puis Police et choisissez « Consolas » ou « Lucida Console » ainsi que la taille, suivant vos envies.

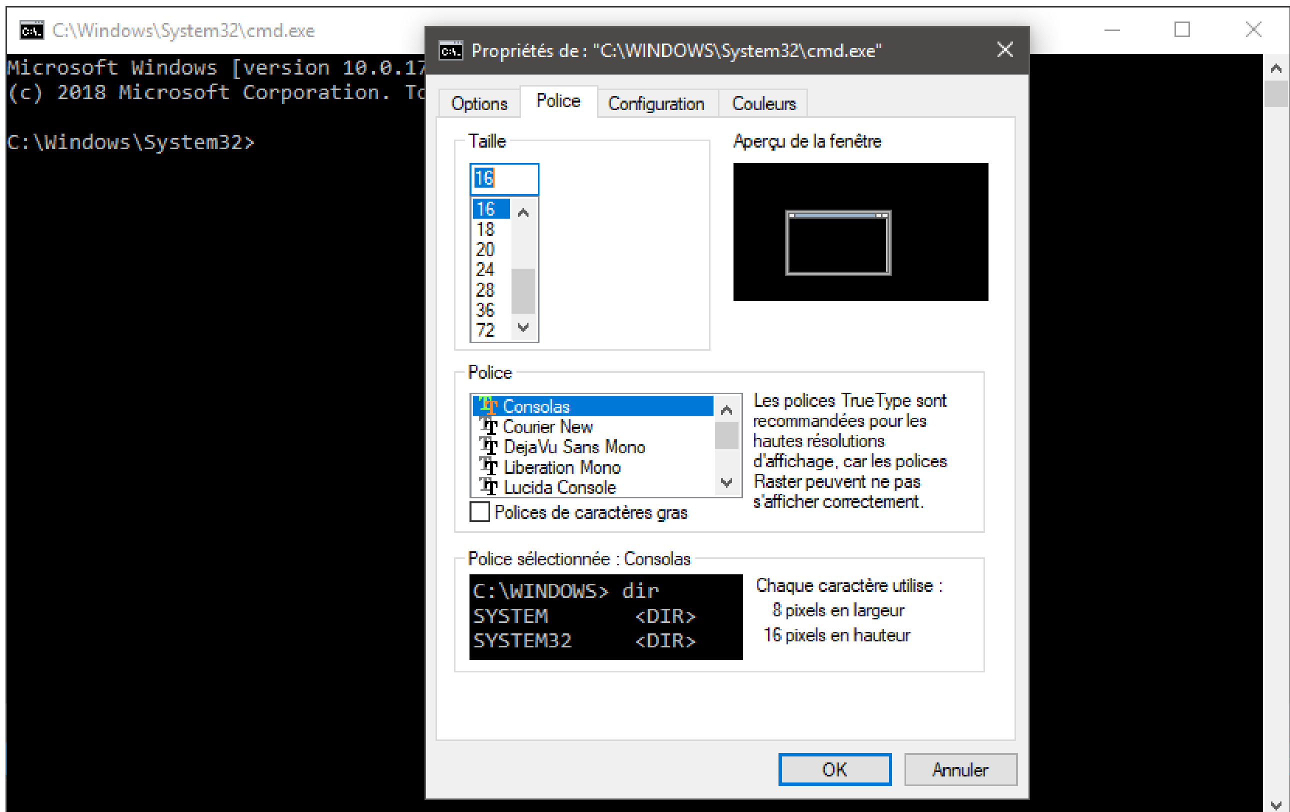


FIGURE 2.10. – Personnellement, j'utilise Consolas en taille 16, que je trouve lisible et agréable.

2.4. Un mot concernant GNU/Linux

Pour nos amis les Linuxiens, il y a Qt Creator, mais ce n'est pas la seule solution. Comme expliqué dans le tutoriel introductif à la programmation, il est également possible d'**utiliser des outils séparés**. La plupart de ceux qui utilisent GNU/Linux au quotidien sont certainement déjà à l'aise avec la console et savent déjà utiliser les outils que je vais présenter. Pour les autres, suivez-moi.

2.4.1. Un éditeur de texte

Ce choix est le votre et peut-être que vous en avez déjà un qui vous convient très bien. Si c'est le cas, très bien, continuez dessus. Sinon, choisissez un éditeur de texte qui au moins possède **la coloration syntaxique**, c'est à dire la capacité de changer la couleur du texte en fonction de ce qu'un mot signifie. Parmi les plus connus, on peut citer Visual Studio Code , Atom ou encore Sublime Text .

2.4.2. Un compilateur

Ici, le choix est plus limité. Les deux compilateurs C++ dont vous entendrez le plus souvent parler, parce qu'ils sont de qualité et à jour, sont GCC et Clang . Pour la version C++ de ces compilateurs, il faut installer soit `g++`, soit `clang++`. Je vous laisse faire, en fonction des commandes d'installation de paquets de votre distribution préférée.

Une fois ceci fait, reste à vous montrer **comment compiler un code C++**. La commande suivante est illustrée avec GCC, mais si vous utilisez Clang, il n'y a qu'à changer le compilateur, le reste étant identique.

```
1 g++ -std=c++17 fichier.cpp -o fichier.out
```

La commande demande à GCC de compiler le fichier `fichier.cpp` et de générer le programme final avec `fichier.out` comme nom. L'option `-std=c++17` indique au compilateur qu'on veut utiliser la version 2017 de C++.

Ne reste plus qu'à écrire un code pour le compiler, ce que nous allons justement voir dans le chapitre suivant.

2.4.3. En résumé

- Nous avons la possibilité de faire des tests très rapidement avec des outils en ligne, qui ne demandent rien d'autre qu'une connexion Internet.
- Nous avons aussi des outils très complets qui nous facilitent la vie, comme Visual Studio 2017 ou Qt Creator. Ils peuvent être utilisés sans connexion Internet et font partie des outils les plus utilisés professionnellement, donc vous gagnez à apprendre à les utiliser.
- Pour que la console Windows soit capable d'afficher les accents, il ne faut pas oublier de changer la police.
- Concernant GNU/Linux, si l'on dispose d'un choix énorme d'éditeurs de texte, les deux grands compilateurs majoritairement utilisés aujourd'hui pour C++ sont GCC et Clang.

3. Rencontre avec le C++

Que diriez-vous de rentrer directement dans le vif du sujet **en écrivant votre premier code C++** ? Nous allons commencer par l'un des exemples les plus connus et les plus utilisés : **afficher un message**. Et, vous verrez, il y a déjà de quoi dire.

3.1. Compilons notre premier programme

Allez donc sur Wandbox , ou bien ouvrez votre projet C++, et copiez-collez ce code dans la zone de texte.

```
#include <iostream>

int main()
{
    std::cout << "Hello World !" << std::endl; // affiche Hello
        World !
    return 0;
}
```

Cliquez ensuite sur Run (pour WandBox), faites Ctrl+F5 (Visual Studio Community), Ctrl+R (Qt Creator), ou compilez et exécutez en ligne de commande, et sous vos yeux émerveillés s'affiche le résultat suivant.

```
Hello World !
```

Félicitations, vous venez de **compiler votre tout premier programme C++** ! Mais je devine que vous voulez comprendre ce qui s'est passé et avoir des explications sur ce que signifie ce code.

3.2. Démystification du code

3.2.1. Inclure des fonctionnalités déjà codées

```
#include <iostream>
```

Le but de notre programme est d'afficher un message. Des développeurs experts ont déjà créé un outil qui permet de le faire facilement. Il se trouve dans un fichier nommé « *iostream* », acronyme de « *Input Output Stream* », soit « Flux d'Entrées Sorties » dans la langue de Molière. Ce fichier fait partie de la **bibliothèque standard C++**, un ensemble de fonctionnalité déjà pré-codées et incluses partout avec chaque compilateur C++.

Pour utiliser les fonctionnalités offertes par ce fichier, notamment écrire un message, on doit l'importer dans notre programme. On dit qu'on l'inclut, d'où l'anglais « *include* ». Nous utiliserons beaucoup cette fonctionnalité tout au long du cours.

Enfin, dernière remarque, avez-vous noté que la ligne commence par le symbole `#` ? C'est ce qu'on appelle une **directive de préprocesseur**. Le préprocesseur est un programme qui se lance automatiquement au début de la compilation, notamment pour importer les fichiers qu'on lui demande. Nous le détaillerons plus loin dans ce cours. Retenez que **`#include` nous permet d'importer des fichiers** que des programmeurs experts ont déjà écrits avant nous.

3.2.2. Le point d'entrée

```
int main()
{

}
```

Quant l'ordinateur exécute un programme, il lui faut bien un endroit où commencer, un début, un point d'entrée. À partir de ce point d'entrée, **il exécutera des instructions**, **des ordres**, que nous aurons au préalable écrits. Cet endroit s'appelle `main` (prononcé « *mèïne* ») et est la fonction principale de tout programme C++.

Qu'est-ce qu'une fonction ? C'est un **un regroupement d'instructions**. En C++, vous trouverez ainsi des fonctions pour calculer une racine carrée, pour comparer des durées, trier une liste d'objets, afficher des messages, etc. Nous en utiliserons beaucoup déjà existantes et, plus tard, apprendrons à créer nos propres fonctions.

Dans notre cas, la fonction `main` regroupe deux instructions différentes.

3.2.3. Voici mes instructions...

```cpp
std::cout << "Hello World !" << std::endl; // Affiche le message
    Hello World !
```

La première instruction est justement celle qui nous permet d'afficher un message. Elle débute avec `std::cout` (à prononcer « ci-aoûte »), qui représente **la sortie standard en C++**, c'est-à-dire la communication du programme vers l'utilisateur, par l'écran dans notre cas. Le préfixe `std::` indique que vous utilisez une fonctionnalité déjà programmée et fournie dans la bibliothèque standard C++.

On note ensuite la présence de deux chevrons `<` qui permettent **d'injecter, d'envoyer, du contenu dans la sortie standard**, dans `std::cout`. Dans notre code d'introduction, nous injectons deux choses.

— D'abord, `"Hello World !"`, qui est le **message que l'on voit s'afficher** à l'écran.
— Puis `std::endl`, qui permet d'**aller à la ligne**, comme son nom l'indique (« *end line* » que l'on traduit par « retour à la ligne »). De même que pour `std::cout`, `std::endl` est fourni par la bibliothèque standard.

Enfin, elle se termine par un point-virgule `;` qui est **obligatoire**. Chaque instruction doit être terminée par un point-virgule. Ce sont les règles du C++ qui veulent ça.

> **i Chevrons**
>
> On peut combiner les chevrons `<` autant que l'on souhaite. Essayez donc d'afficher la phrase « Bonjour, j'écris du C++ » suivi d'un retour à la ligne, suivi de « Et j'aime beaucoup. »

Contenu masqué n°1

> **! Un mot sur Windows**
>
> Si vous êtes sous Windows et que vous tentez d'afficher du texte avec des accents, vous allez obtenir des caractères bizarres. Pour afficher du code avec des accents, vous allez devoir ajouter les lignes 2, 3 et 8 dans votre code. Nous en reparlerons plus tard.

!

```
#include <iostream>
#define NOMINMAX
#include <Windows.h>

int main()
{
    // À utiliser avant d'afficher du texte.
    SetConsoleOutputCP(1252);
    std::cout << "Oui, je peux utiliser éèàï !" << std::endl;
}
```

Sous GNU/Linux et MacOS, vous n'aurez aucun problème, donc ignorez ces lignes.

```
return 0;
```

La deuxième instruction conclut la fonction `main` en renvoyant la valeur `0`. Le mot anglais pour renvoyer est *return* ; vous croiserez d'ailleurs souvent l'anglicisme retourner.

Mais à qui est transmise cette valeur ? Au système d'exploitation. Le zéro, par convention, signifie que tout s'est bien passé.

Nous aurons l'occasion de comprendre mieux tout cela quand nous aborderons le chapitre sur les fonctions. Ne vous inquiétez donc pas de `return 0;`, **sachez simplement qu'il termine la fonction `main`**.

3.3. Les commentaires

Cela ne vous a pas échappé, mais j'ai omis de traiter d'un bout de code. En effet, cette partie bizarre avec des slashs.

```
// Affiche le message Hello World !
```

C'est ce qu'on appelle **un commentaire**. Les commentaires sont complètement ignorés par le compilateur, **ils ne servent qu'aux humains**. On s'en sert pour documenter son code, expliquer des passages un peu ardus, décrire des passages un peu moins lisibles ou tout simplement pour offrir quelques compléments d'information.

En effet, un code est bien plus souvent lu qu'écrit ou modifié. C'est pourquoi il est important **qu'il soit lisible**. Les commentaires aident à gagner en lisibilité.

Enfin, contrairement au code qui contient beaucoup de mots anglais, les commentaires peuvent être écrits dans la langue que l'on souhaite. Si beaucoup de projets collaboratifs ont des

commentaires écrits en anglais, dans ce tutoriel, je les écrirai en français afin de vous aider à bien comprendre.

3.3.1. La syntaxe

On peut écrire des commentaires de deux façons. La première, c'est celle que nous avons vu avec les deux slashs `//`. Un commentaire ainsi créé s'arrête à la fin de la ligne où il est créé. On dit que c'est un **commentaire mono-ligne**.

```
// Cette ligne affiche Bonjour.
std::cout << "Bonjour";

// Cette ligne permet d'afficher un retour à la ligne.
std::cout << std::endl;
```

L'autre syntaxe permet d'écrire des blocs de commentaires bien délimités entre les symboles `/*` et `*/`. On peut ainsi écrire des commentaires sur plusieurs lignes, voire entre plusieurs instructions !

```
/* Ceci est un commentaire sur une seule ligne. */
std::cout << "Hello World !" << std::endl; /* Ceci est un
    commentaire à la fin d'une ligne du code. */

/* Ceci...
...est un commentaire
écrit sur plusieurs lignes...
...car il est très long. */

std::/* Ceci est un commentaire noyé dans le code. */cout <<
    "Hello World !";
```

Vu que les commentaires sont ignorés par le compilateur, on peut tout à fait en écrire en plein milieu d'une ligne de code. Mais ce n'est pas lisible, aussi je vous déconseille fortement de le faire, car un code est, je vous le rappelle, plus souvent lu qu'écrit.

3.3.2. En résumé

- Un programme en C++ est composé d'instructions, qui sont les ordres que l'on donne à l'ordinateur.
- Ces instructions sont regroupées dans des blocs appelés fonctions. Celle par laquelle commence tout programme C++ s'appelle `main`.
- Nous avons utilisé la bibliothèque standard pour la première fois, avec le fichier `iostream`, afin d'afficher un message à l'écran.

— Les commentaires sont ignorés par le compilateur mais nous sont très utiles à nous, humains.

Contenu masqué

Contenu masqué n°1

```cpp
#include <iostream>

int main()
{
    std::cout << "Bonjour, j'écris du C++" << std::endl <<
        "Et j'aime beaucoup.";

    return 0;
}
```

Retourner au texte.

4. Une super mémoire

Vous savez désormais afficher des messages à l'écran. C'est un bon début, mais on est quand même assez limités, n'est-ce pas ? Rassurez-vous, C++ nous permet de faire bien plus que ça. Grâce à ce langage, nous allons nous amuser avec cette **super calculatrice** qu'est notre ordinateur et nous découvrirons qu'il possède aussi **une super mémoire** !

Ce chapitre va nous faire découvrir les **littéraux** et les **variables**.

4.1. Les littéraux

Que sont les littéraux ? **Une valeur écrite littéralement dans le code**, d'où son nom. Nous avons rencontré un littéral dans le chapitre précédent : `"Hello World !"`. Ce littéral est ce qu'on appelle une **chaîne de caractères**. Mais ce n'est pas le seule littéral possible en C++. Examinons tout ça.

4.1.1. Les caractères

Nous avons fait connaissance avec les chaînes de caractères lors du chapitre précédent. Les chaînes de caractères ne sont rien d'autre que du texte. On les reconnait parce qu'elles commencent et finissent par des doubles guillemets `"`.

Entraînez-vous donc à afficher des chaînes de caractères, par exemple un littéral souhaitant une bonne journée.

Contenu masqué n°2

Tous comme les mots de la langue française sont des regroupements de lettres, les chaînes de caractères sont une suite de **caractères simples**. En C++, un caractère est encadré par des guillemets simples `'`.

```
1 #include <iostream>
2
3 int main()
4 {
5     // Un caractère peut être une lettre.
6     std::cout << 'A' << std::endl;
7     // Ou bien un chiffre.
```

```
8        std::cout << '7' << std::endl;
9        // Ou même de la ponctuation.
10       std::cout << '!' << std::endl;
11
12       return 0;
13   }
```

Les chaînes de caractères permettent de regrouper des caractères et nous facilitent ainsi la vie pour écrire du texte.

```
1   #include <iostream>
2
3   int main()
4   {
5       // Que c'est fastidieux d'écrire ça !
6       std::cout << 'S' << 'a' << 'l' << 'u' << 't' << ' ' << 't' <<
            'o' << 'i' << ' ' << '!' << std::endl;
7       // C'est tellement mieux comme ça.
8       std::cout << "Salut toi !" << std::endl;
9
10      return 0;
11  }
```

?

J'ai mis plusieurs caractères entre guillemets simples, comme ceci `'abc'` et mon code compile. Ça veut dire que ça marche ?

Si vous avez essayé de compiler, vous avez remarqué un message en rouge dans la fenêtre de résultat disant `warning: multi-character character constant`, ainsi qu'un nombre au lieu de nos caractères.

Le ***warning*** est un message d'avertissement que le compilateur vous envoie, car il ne sait pas si ce que vous avez fait est une erreur d'inattention ou bien une manœuvre volontaire de votre part. Dans notre cas, la norme C++ n'interdit pas de faire ça, mais **ce n'est pas considéré comme une bonne pratique**, comme étant du bon code.

!

N'ignorez pas les warnings !

Les *warnings* sont des messages importants signalant d'éventuels problèmes. Il ne faut surtout pas les ignorer sous prétexte que le code compile !

4.1.1.1. Des caractères spéciaux

Il existe quelques caractères qui sont un peu particuliers et, pour les introduire, je vais vous demander d'afficher un message contenant un chemin de dossier Windows (`C:\Program Files`

`(x86)` par exemple).

Contenu masqué n°3

Boum ba da boum! Le compilateur vous crache à la figure un message de type `warning: unknown escape sequence: '\P'`. Que s'est-il passé ?

Il existe en C++ des séries de caractères, appelées **séquences d'échappement**, qui commencent toutes par `\` et dont la liste complète se trouve ici . Et pour comprendre l'intérêt de ces séquences d'échappement, essayez donc de compiler le code suivant.

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Il m'a demandé " Comment vas-tu ? "" <<
           std::endl;
6
7      return 0;
8  }
```

Vous devez certainement avoir une erreur proche de celle-ci : `'error: expected ';' before 'Comment'`. C'est tout à fait normal. En effet, nous avons vu plus haut que les chaînes de caractères sont délimitées par les doubles guillemets `""`. Donc tout ce qui est entre cette paire de guillemets est considéré par le compilateur comme faisant partie de la chaîne de caractères.

Dans notre exemple, cela veut dire que `Comment vas-tu ?` **ne fait plus partie** de la chaîne. Donc le compilateur l'interprète comme des instructions C++ et comme il ne les comprend pas, il ne peut pas compiler le programme.

Mais quel rapport avec ces fameux caractères commençant par `\` ?

Eh bien, les séquences d'échappement permettent de dire au compilateur « Écoute, ce caractère est spécial, il faut l'afficher et non l'interpréter », ce qui permet d'afficher des doubles guillemets dans une chaîne de caractères par exemple.

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Il m'a demandé \" Comment vas-tu ? \"" <<
           std::endl;
6      std::cout << "Dossier principal : C:\\Program Files (x86)" <<
           std::endl;
```

```
7
8     return 0;
9 }
```

De toutes les séquences d'échappement qui existe, les plus utilisées et celles que vous verrez le plus souvent sont les suivantes.

— `\'` qui permet d'afficher un guillemet simple `'`.
— `\"` qui permet d'afficher un guillemet double `"`.
— `\n` qui permet d'aller à la ligne, comme `std::endl`.
— `\t` qui permet de faire une tabulation horizontale.
— `\\` qui permet d'afficher un antislash `\`.

Entraînez-vous donc en affichant un guillemet simple sans échappement, puis avec échappement. Et faites donc mumuse avec les tabulations et les retours à la ligne.

Contenu masqué n°4

4.1.2. Les nombres

Il existe d'autres types de littéraux en C++ : les nombres entiers et les nombres à virgule, appelés **flottants**.

```
1 std::cout << -1 << std::endl;
2 std::cout << 0 << std::endl;
3 std::cout << 1 << std::endl;
4
5 std::cout << -1.6027 << std::endl;
6 std::cout << 3.14159 << std::endl;
7 std::cout << 2.71828 << std::endl;
```

On remarque déjà qu'on peut utiliser des nombres négatifs sans aucun problème. Ensuite, dû à l'origine américaine de C++, **les flottants s'écrivent avec un point et non une virgule**. Même si l'on se fiche des chiffres après la virgule, il faut mettre le point `.` sinon C++ interprétera le nombre comme un entier.

Sinon, rien de bien étonnant, ce sont des nombres et on peut faire des opérations dessus.

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "1 + 2 = " << 1 + 2 << std::endl;
```

```
6       std::cout << "1 - 4 = " << 1 - 4 << std::endl;
7       std::cout << "7.5 * 2 = " << 7.5 * 2 << std::endl;
8
9       std::cout << "9 / 2 = " << 9 / 2 << std::endl;
10      std::cout << "9. / 2 = " << 9. / 2 << std::endl;
11      std::cout << "9 % 2 = " << 9 % 2 << std::endl;
12
13      return 0;
14  }
```

?

Attends, qu'est-ce que c'est que ça ? `9 / 2` et `9. / 2` ne donne pas la même chose ? Et c'est quoi `%` ?

En fait, la raison est très simple : pour C++, si on fait une opération sur deux nombres entiers, le résultat est un nombre entier. Si l'on veut que le résultat soit un nombre à virgule, il faut qu'au moins un des deux nombres soit un flottant.

Dans le cas de la division, si les deux nombres sont des entiers, on obtient le quotient de la division euclidienne, le reste s'obtenant avec l'opérateur `%`, appelé **modulo** (c'est de l'arithmétique). Si l'un des deux nombres est un flottant, alors on obtient le résultat de la division réelle, donc un nombre réel.

Également, comme en maths, les calculs respectent la distributivité , l'associativité , la commutativité et la priorité des opérateurs . L'exemple suivant est tiré du cours de @gbdivers.

```
1   #include <iostream>
2
3   int main()
4   {
5       std::cout << "Commutativité :" << std::endl;
6       std::cout << "2 + 3 = " << 2 + 3 << std::endl;
7       std::cout << "3 + 2 = " << 3 + 2 << std::endl;
8
9       std::cout << "Associativité :" << std::endl;
10      std::cout << "2 + (3 + 4) = " << 2 + (3 + 4) << std::endl;
11      std::cout << "(2 + 3) + 4 = " << (2 + 3) + 4 << std::endl;
12
13      std::cout << "Distributivité :" << std::endl;
14      std::cout << "2 * (4 + 3) = " << 2 * (4 + 3) << std::endl;
15      std::cout << "2 * 4 + 2 * 3 = " << 2 * 4 + 2 * 3 << std::endl;
16
17      std::cout << "Priorité des opérateurs :" << std::endl;
18      std::cout << "2 * 6 + 3 = " << 2 * 6 + 3 << std::endl;
19      std::cout << "2 * (6 + 3) = " << 2 * (6 + 3) << std::endl;
20      std::cout << "2 * 4 - 6 / 2 = " << 2 * 4 - 6 / 2 << std::endl;
```

```
21
22      return 0;
23  }
```

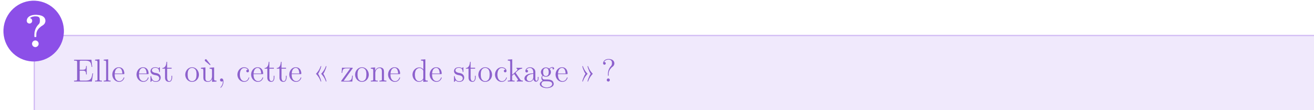
Nous nous sommes amusés avec les littéraux, mais on est quand même rapidement limité. Imaginons que nous voulions multiplier le résultat précédent par 2 et ce, trois fois. Nous sommes obligés d'écrire un code comme ceci.

```
1   #include <iostream>
2
3   int main()
4   {
5       std::cout << "1 * 2 = " << 1 * 2 << std::endl;
6       std::cout << "1 * 2 * 2 = " << 1 * 2 * 2 << std::endl;
7       std::cout << "1 * 2 * 2  * 2 = " << 1 * 2 * 2 * 2 << std::endl;
8
9       return 0;
10  }
```

Maintenant, imaginons qu'on ne multiplie plus par 2 mais par 4, et non plus trois fois mais cinq. Vous visualisez bien tous les changements à faire ? C'est pénible, n'est-ce pas ? N'y a-t-il pas un moyen de se souvenir de valeurs et de calculs ?

4.2. Les variables

La réponse à la question soulevée dans la section précédente se trouve dans le titre. Il s'agit des **variables**. C'est un concept commun à beaucoup de langages de programmation qui permet de **stocker une valeur** et de **lui associer un nom**, afin de faciliter tant l'écriture que la lecture du code. On peut les voir comme des enveloppes, des tiroirs, des conteneurs, bref, une zone où est stockée une valeur, à laquelle on associe un nom.


Elle est où, cette « zone de stockage » ?

Toutes les variables sont stockées dans la mémoire vive de l'ordinateur, la RAM . Son fonctionnement est un peu complexe et ne nous intéresse pas (dans le cadre de ce cours tout du moins). Les curieux trouveront beaucoup de ressources sur Internet pour combler leur soif de savoir, allant de la vulgarisation simple à des explications complètes .

4.2.1. Comment créer des variables en C++ ?

Pour déclarer une variable en C++, il faut trois éléments obligatoires.

— D'abord, un **type**, qui indique ce que la variable va stocker (un entier, une chaîne de caractères, etc).
— Ensuite, un **identificateur**, c'est-à-dire le nom associé à la variable.
— Enfin, il faut bien donner **une valeur** à stocker à notre variable. Ceci se fait en mettant cette valeur entre accolades `{ }`.

Examinons un cas réel que nous détaillerons.

```cpp
#include <iostream>
#include <string>

int main()
{
    int reponse { 42 };
    std::cout << "La réponse à la Grande Question est " << reponse
        << std::endl;

    double pi { 3.1415926 };
    std::cout << "Voici la valeur du célèbre nombre pi : " << pi <<
        std::endl;

    char lettre { 'A' };
    std::cout << "La première lettre de l'alphabet français est "
        << lettre << std::endl;

    std::string phrase { "Bonjour tout le monde !" };
    std::cout << "En entrant dans la salle, il s'écria : " <<
        phrase << std::endl;

    return 0;
}
```

```
La réponse à la Grande Question est 42
Voici la valeur du célèbre nombre pi : 3.14159
La première lettre de l'alphabet français est A
En entrant dans la salle, il s'écria : Bonjour tout le monde !
```

Syntaxe héritée

Il existe une syntaxe alternative, de la forme `type identificateur = valeur;`. Essayez, vous verrez que ça marche.

```cpp
#include <iostream>

int main()
{
    int variable = 42;
    std::cout << variable << std::endl;

    return 0;
}
```

Cette syntaxe est héritée du C. Elle est toujours valable en C++, ne soyez donc pas surpris si vous la voyez un jour dans des codes divers. Dans ce cours, nous utiliserons la forme dite « moderne ».

4.2.1.1. C'est tout à fait mon type

Les plus attentifs ont remarqué que l'on retrouve les mêmes littéraux que dans la section précédente.

- Pour les **nombres entiers**, nous avons le mot-clé **`int`**, abréviation de l'anglais *integer* signifiant … nombre entier. Grâce à ce type, on peut stocker des entiers négatifs ou positifs.
- Pour les **flottants**, les nombres à virgule, nous avons le mot-clé **`double`**, qui permet de stocker des nombres réels très grands.
- Pour les **caractères simples**, nous avons **`char`**, qui est tellement transparent que je ne vous ferai pas l'affront de le traduire.
- Enfin, pour les **chaînes de caractères**, nous avons le type **`std::string`**. Ce type est un peu particulier car il n'existe pas nativement en C++. Ce sont des programmeurs experts qui l'ont codé afin de manipuler aisément des chaînes de caractères. Afin de pouvoir manipuler des `std::string`, il faut donc inclure le bon fichier, ce que l'on fait grâce à la ligne `#include <string>`.

4.2.1.2. Quel est ton nom, petite variable ?

D'autre, puristes de la langue française, auront remarqué qu'il manque l'accent aigu dans l'identificateur `reponse`. Ce n'est pas un oubli de ma part. Si je suis obligé de faire cette entorse à notre belle langue, c'est parce que **C++ m'y force**. Étant un langage inventé aux États-Unis dans les années 1980, **C++ n'autorise que les 26 lettres composant l'alphabet anglais** (plus les chiffres et l'underscore `_`), à savoir donc les mêmes que les nôtres, accents non compris. Dans la réalité, les programmeurs utilisent l'anglais pour nommer leurs variables, ce n'est donc pas un problème.

Langue du cours

Je n'utilise le français que dans le cadre de votre apprentissage. Quand vous aurez pris un peu de bouteille, je vous encouragerai à utiliser l'anglais.

Mais que ce soit en anglais ou en français, un identificateur doit **respecter des règles bien précises**.

— Il doit **commencer par une lettre**. Il ne peut pas commencer par un chiffre, c'est interdit. Il ne doit pas commencer non plus par underscore `_` : leur utilisation répond à des règles précises, donc il est plus simple de ne pas les employer comme premier caractère.
— Les espaces et les signes de ponctuation sont **interdits** (`'`, `?`, etc).
— On **ne peut pas utiliser un mot-clé** du langage comme identificateur. Ainsi, il est interdit de déclarer une variable s'appelant `int` ou `return`, par exemple.

```
#include <iostream>

int main()
{
    // Bon exemple.
    int avec_underscore { 0 };
    // Erreur : espace interdit.
    int avec espace { 0 };

    // Bon exemple.
    int variable1 { 42 };
    // Erreur : ne peut pas commencer par un chiffre.
    int 1variable { 42 };

    // Bon exemple.
    char lettre { 'A' };
    // Erreur : ponctuation interdite.
    char autre_lettre! { 'B' };

    // Bon exemple.
    double retour { 2.71 };
    // Erreur : mot-clé réservé par C++.
    double return { 2.71 };

    return 0;
}
```

4.2.1.3. De l'importance d'un beau nom

Donner un nom clair et précis à une variable est un grand défi auquel sont confrontés même les programmeurs ayant de l'expérience. En effet, même si un identificateur respecte les règles

imposées par C++, **cela ne veut pas dire que la variable est bien nommée**.

— `variable` : ne veut rien dire. Que stocke-t-elle ? Quel est son but ? Pourquoi existe-t-elle ?
— `variable_contenant_la_multiplication_de_pi_par_e_au_cube` : beaucoup trop long.
— `cIrCoNfErEnCe_CeRcLe` : le nom définit bien le pourquoi de l'existence de notre variable, mais le mélange majuscule / minuscule le rend illisible.
— `Brzęczyszczykiewicz` : aucun sens, sauf si vous aimez l'humour polonais .

Avec le temps, en progressant, vous arriverez à trouver plus rapidement et plus facilement des identificateurs qui soient clairs et simples. Dans le cours, je m'efforcerai de le faire afin de vous donner des exemples concrets.

4.2.1.4. Sinon, par défaut...

Abordons un dernier point. Je vous ai dit qu'on donne une valeur entre accolades pour initialiser notre variable. Mais que se passe t-il si on écrit simplement les accolades, sans aucune valeur dedans ? Le code va t-il toujours compiler ?

```
#include <iostream>
#include <string>

int main()
{
    int entier {};
    std::cout << "Mon entier vaut " << entier << std::endl;

    double reel {};
    std::cout << "Mon réel vaut " << reel << std::endl;

    char lettre {};
    std::cout << "Mon caractère vaut " << lettre << std::endl;

    std::string phrase {};
    std::cout << "Ma chaîne vaut " << phrase << std::endl;

    return 0;
}
```

```
Mon entier vaut 0
Mon réel vaut 0
Mon caractère vaut
Ma chaîne vaut
```

Non, le programme n'a pas de *bugs*, il fonctionne très bien. En fait, si aucune valeur n'est spécifiée, nos variables sont **initialisées à une valeur par défaut**. Pour les entiers et les réels,

il s'agit de zéro. Pour les caractères, c'est une valeur spéciale signifiant « pas de caractère », de même que pour les chaînes de caractères, initialisée avec du vide.

À vous de choisir

C++ vous laisse libre de vos choix. Dans ce cours, pour commencer, nous écrirons toujours les valeurs explicitement. Au fur et à mesure que vous progresserez, nous utiliserons de plus en plus souvent l'initialisation par défaut.

4.2.2. Un peu de constance, voyons !

Terminons cette partie en présentant un mot-clé important qui a pour nom `const`. Il permet d'empêcher toute modification de la variable sur laquelle ce mot-clé s'applique. On dit qu'on crée **une constante**. Et si l'on essaye de modifier une constante, le compilateur refuse clair et net.

```cpp
int main()
{
    double const pi { 3.141592 };
    pi = 4; // Ouch, badaboum ça ne compile pas.

    return 0;
}
```

```
[Visual Studio] C3892 'pi' : vous ne pouvez pas assigner une
    variable const.

[GCC] prog.cc: In function 'int main()':
prog.cc:4:8: error: assignment of read-only variable 'pi'
    4 |     pi = 4; // Ouch, badaboum ça ne compile pas.
      |     ~~~^~~

[Clang] prog.cc:4:8: error: cannot assign to variable 'pi' with
    const-qualified type 'const double'
    pi = 4; // Ouch, badaboum ça ne compile pas.
    ~~ ^
prog.cc:3:18: note: variable 'pi' declared const here
    double const pi { 3.141592 };
    ~~~~~~~~~~~~~^~~~~~~~~~~~~~
1 error generated.
```

Certains choisissent de les écrire entièrement en majuscule pour bien les différencier des autres variables. Ce n'est absolument pas une obligation et nous n'imposons sur ce point aucune règle.

Quelle en est l'utilité ? Il sert pour la qualité du code. Tout ce qui n'est pas destiné à être modifié est ainsi protégé. Cela nous permet également de donner des garanties fortes à notre code. C'est pour ça que nous allons l'utiliser dans tous les exemples. Vous verrez plus tard dans le cours d'autres cas utiles.

Ordre du `const`

Vous verrez souvent des codes qui inversent l'ordre de `const` et écrivent `const int constante {};`. Cela est légal et possible en C++, car `const` respecte une règle simple : **il s'applique à ce qui est à sa gauche immédiate**, sauf s'il n'y a rien, auquel cas il s'applique à **ce qu'il y a à droite**. Ici, cela ne change rien, mais plus tard cela aura des conséquences. Nous verrons cela en temps voulu.

4.2.3. Manipulation de variables

Nous ne sommes absolument pas obligés ne nous contenter d'afficher nos variables. Au contraire, nous pouvons faire beaucoup d'opérations dessus, notamment **combiner les variables entre elles** et **modifier leur valeur**.

```
1  #include <iostream>
2
3  int main()
4  {
5      int entier { 4 };
6      std::cout << "Mon entier vaut : " << entier << std::endl;
7
8      // Je peux tout à fait changer la valeur de ma variable.
9      entier = 4 * (8 + 9) - 1;
10     std::cout << "Finalement non, il vaut : " << entier <<
           std::endl;
11
12     // Je peux même utiliser la valeur de ma variable et la
           réaffecter à la même variable.
13     entier = entier + 7;
14     std::cout << "Et si j'additionne 7 ? " << entier << std::endl;
15
16     int autre_entier { entier * 2 };
17     // Je peux utiliser d'autres variables également.
18     entier = autre_entier - 4;
19     std::cout <<
           "Finalement, je décide que mon entier a une autre valeur qui est : "
           << entier << std::endl;
20
21     return 0;
22 }
```

Pour modifier la valeur d'une variable, on utilise l'opérateur d'affectation `=`, précédé de l'identificateur de la variable et suivi de la valeur à affecter : `identificateur = valeur;`. Et comme vous l'avez noté, cette valeur peut être un simple littéral, un calcul plus complexe, une autre variable voire la variable actuelle elle-même.

En effet, C++ s'occupe d'abord de tout ce qui se trouve **à droite du signe égal**. Ainsi, dans la ligne `entier = entier + 7`, il récupère l'ancienne valeur de `entier`, lui ajoute 7 et assigne ce nouveau résultat à `entier`. Essayez donc pour voir.

```cpp
#include <iostream>

int main()
{
    int entier { 4 };
    std::cout << "Mon entier vaut : " << entier << std::endl;

    entier = entier + 2;
    std::cout << "Mon entier vaut : " << entier << std::endl;

    entier = entier - 2;
    std::cout << "Mon entier vaut : " << entier << std::endl;

    entier = entier * 2;
    std::cout << "Mon entier vaut : " << entier << std::endl;

    entier = entier / 2;
    std::cout << "Mon entier vaut : " << entier << std::endl;

    return 0;
}
```

Vous souvenez-vous de l'exemple que nous avions vu à la fin de la section sur les littéraux ? Nous faisions des multiplications, mais le fait de devoir répéter de nombreuses fois les mêmes instructions était lourd. Tenez, entraînez-vous donc en faisant un programme qui utilise les variables.

Contenu masqué n°5

4.2.3.1. Prenons un raccourci

Les programmeurs étant fainéants par nature, ils n'aiment pas la répétition. Ils ont donc créé des raccourcis pour aller plus vite. Ainsi, plutôt que d'écrire `a = a + b`, on peut écrire directement `a += b`. Cette nouvelle syntaxe s'applique aux opérateurs d'addition `+`, de soustraction `-`, de multiplication `*`, de division `/` et de modulo `%`.

```cpp
#include <iostream>

int main()
{
    int entier { 4 };
    std::cout << "Mon entier vaut : " << entier << std::endl;

    entier += 2;
    std::cout << "Mon entier vaut : " << entier << std::endl;

    entier -= 2;
    std::cout << "Mon entier vaut : " << entier << std::endl;

    entier *= 2;
    std::cout << "Mon entier vaut : " << entier << std::endl;

    entier /= 2;
    std::cout << "Mon entier vaut : " << entier << std::endl;

    entier %= 2;
    std::cout << "Mon entier vaut : " << entier << std::endl;

    return 0;
}
```

Pire, non contents de ça, ils ont raccourci encore la syntaxe dans le cas où l'on souhaite ajouter ou soustraire 1 à un nombre. On parle d'**incrémentation** et de **décrémentation**. Il suffit de faire ainsi.

— `a++` ou `++a` pour incrémenter de 1 la valeur de la variable.
— `a--` ou `--a` pour décrémenter de 1 la valeur de la variable.

Les deux syntaxes sont quasiment équivalentes, subsiste une différence mineure que je passe sous silence car elle ne nous intéresse pas à l'heure actuelle.

4.3. Qu'en déduisez-vous ?

Lorsque nous déclarons une variable, comme vous le savez maintenant, nous précisons, entre autres, son type et sa valeur. Sauf que cette information est parfois redondante. Regardez le code d'illustration ci-dessous.

```cpp
int main()
{
    // Comme 0 est déjà un littéral entier, pourquoi donc préciser
        'int' ?
    int entier { 0 };
```

```
5
6       // Comme 3.1415 est déjà un littéral flottant, pourquoi donc
            préciser 'double' ?
7       double reel { 3.1415 };
8
9       return 0;
10  }
```

Le compilateur peut en effet déduire tout seul le type d'une variable et cela grâce à un nouveau mot-clef : `auto`. Celui-ci s'utilise en lieu et place du type de la variable. Notre code précédent peut ainsi être réécrit de la manière qui suit.

```
1   int main()
2   {
3       // Hop, le compilateur déduit que 'entier' est de type 'int'.
4       auto entier { 0 };
5
6       // Hop, le compilateur déduit que 'reel' est de type 'double'.
7       auto reel { 3.1415 };
8
9       return 0;
10  }
```

4.3.1. Avec `const`

Tout comme dans le cas d'une variable dont le type est explicitement écrit, si vous voulez déclarer une constante, il suffit de rajouter `const`.

```
1   int main()
2   {
3       auto const constante { 40 };
4       return 0;
5   }
```

4.3.2. Le cas de `std::string`

Dans le cas des chaînes de caractères, c'est un peu particulier. Déjà, il faut rajouter un `s` après les guillemets fermants `"`, et ce pour chaque chaîne de caractères que vous voulez écrire. Ensuite, il faut rajouter une ligne spéciale. Nous expliquerons pourquoi plus loin dans ce cours. L'exemple suivant va vous aider à comprendre.

```
#include <string>

int main()
{
    // Écrivez cette ligne une seule fois.
    using namespace std::literals;

    // Puis vous pouvez déclarer autant de chaînes de caractères
       que vous voulez.
    auto chaine { "Du texte."s };
    auto autre_chaine { "Texte alternatif."s };
    auto encore_une_chaine { "Allez, un dernier pour la route."s };

    return 0;
}
```

i

Histoire et héritage

Si nous sommes obligés de rajouter le `s` en fin de littéral, c'est parce que nous avons affaire ici à un **héritage du C**. Sans celui-ci, le compilateur déduit que notre littéral est une chaîne de caractères comme en C et non comme étant une `std::string`. Nous en reparlerons au moment voulu, mais retenez que ce `s` est **très important** dès que vous utilisez `auto`.

4.3.3. Les avantages

Utiliser `auto` présente deux avantages majeurs.

— D'abord, utiliser `auto` permet de **raccourcir le code**. Là, remplacer `double` par `auto` n'est pas un gain de place énorme, mais, plus tard dans ce cours, nous verrons des types longs à écrire et lourds à lire, qu'il sera bien plus agréable de remplacer par un concis `auto`.

— Ensuite, cela permet un **code plus évolutif**. Imaginez des calculs complexes avec des types écrits explicitement. Si vous voulez changer l'un des types, il faut en changer chaque occurrence. Avec `auto`, vous laissez au compilateur le soin de s'occuper de cette tâche.

?

Je suis perdu. Qu'est-ce que je dois choisir entre `auto` ou le type explicite ?

C++ vous autorise à faire les deux, donc **le choix est vôtre**. Certains programmeurs ne jurent que par `auto` et l'utilisent partout. D'autres préfèrent le réserver aux types longs ou complexes et écrire explicitement les types simples et courts. Dans le cadre de ce cours, j'utiliserai les deux façons de faire.

4.4. Les entrées

Dès le premier code C++ que vous avez étudié, il y avait la notion de sortie standard. Il est temps de voir le concept inverse en manipulant **l'entrée standard**. Maintenant que nous connaissons le concept des variables, demander des informations à l'utilisateur est à notre portée. Il faut en effet pouvoir stocker quelque part la valeur qu'a tapée l'utilisateur, ce que permettent justement les variables.

Manipuler les entrées est un peu particulier avec Wandbox. Il faut cliquer sur l'encadré `stdin` et taper la valeur que vous souhaitez rentrer. C'est moins pratique qu'avec un IDE. Peu importe ce que vous préférez, l'essentiel est que vous puissiez manipuler pour comprendre la suite du chapitre.

```cpp
#include <iostream>

int main()
{
        std::cout << "Entre ton age : " << std::endl;
        int age { 0 };
        std::cin >> age;
        std::cout << "Tu as " << age << " ans.\n";

        return 0;
}
```

Avez-vous remarqué à quel point `cin` est semblable à `cout` ? Déjà, pour l'utiliser, il faut le préfixer par `std::`, car `cin` est un objet appartement à la bibliothèque standard. Et on utilise les chevrons dans le sens inverse de `cout`.

4.4.1. Gestion des erreurs

Et si vous rentrez une valeur qui n'a rien à voir ? Que se passe-t-il si je décide de taper `Maurice` au lieu de `5` ? Essayez donc.

```
Entre ton age :
Maurice
Tu as 0 ans.
Appuyez sur une touche pour continuer...
```

Comme on veut récupérer un entier et qu'on a écrit une chaîne de caractères, notre variable `age` n'est pas modifiée et vaut donc toujours `0`, sa valeur par défaut.

? Donc c'est bon alors ? Y'a pas de soucis particulier, puisque `std::cin` ignore nos bêtises.

Eh bien nous allons voir que si. Essayons donc de demander deux informations à l'utilisateur : son âge et son nom.

```
#include <iostream>
#include <string>

int main()
{
        std::cout << "Entre ton age : ";
        int age { 0 };
        std::cin >> age;
        std::cout << "Tu as " << age << " ans.\n";

        std::cout << "Entre ton nom : ";
        std::string nom { "" };
        std::cin >> nom;
        std::cout << "Tu t'appelles " << nom << ".\n";

        return 0;
}
```

Lancez ce code en faisant exprès de taper une chaîne de caractères plutôt qu'un entier. Vous devriez obtenir le même résultat que si dessous.

```
Entre ton age : Mishka
Tu as 0 ans.
Entre ton nom : Tu t'appelle .
Appuyez sur une touche pour continuer...
```

Vous n'avez rien eu le temps de taper que le programme avait fini. C'est parce que, lorsque `std::cin` rencontre une erreur (comme le fait de rentrer du texte alors qu'on attend un entier), **il passe dans un état invalide**, tous les caractères invalides restent mémorisés et toutes les utilisations suivantes de `std::cin` sont foireuses.

Comment allons-nous gérer ça ? Comment nettoyer `std::cin` s'il a échoué ? Le chapitre suivant vous apprendra un mécanisme basique mais puissant, qu'on trouve tout le temps en programmation et qui nous aidera à régler nos problèmes.

> **!** Un mot sur Windows
>
> La gestion des accents et autres sous Windows, avec C++, est compliquée. Malheureusement, la seule solution que j'ai à vous offrir est **de vous passer des accents** et autres caractères français spéciaux. Tant pis pour la beauté de la langue française.
> Dans le cadre professionnel, l'anglais est majoritairement utilisé, donc il n'y a pas de problème.

4.4.2. En résumé

- C++ nous permet de manipuler des caractères simples, des chaînes de caractères, des nombres entiers et des nombres réels.
- Il existe des caractères spéciaux, comme le retour à la ligne (`'\n'`) ou le caractère nul (`'\0'`).
- Les variables nous permettent d'associer un nom à une valeur, mais il faut respecter certaines règles.
- On peut utiliser `auto` pour laisser le compilateur déduire tout seul le type d'une expression.
- Nous pouvons demander des informations à l'utilisateur grâce à `std::cin`.
- Nous n'avons aucun mécanisme de protection s'il rentre n'importe quoi.

Contenu masqué

Contenu masqué n°2

```cpp
#include <iostream>

int main()
{
    std::cout << "Bonne journée à toi lecteur." << std::endl;
    return 0;
}
```

Retourner au texte.

Contenu masqué n°3

```cpp
#include <iostream>

int main()
{
    std::cout << "Dossier principal : C:\Program Files (x86)" <<
        std::endl;

    return 0;
}
```

Retourner au texte.

Contenu masqué n°4

```cpp
#include <iostream>

int main()
{
    std::cout << "Voici un exemple sans échappement : " << "'" <<
        " est bien affiché." << std::endl;
    std::cout << "Maintenant, je vais l'échapper : " << '\'' <<
        " est bien affiché grâce à l'échappement." << std::endl;

    std::cout << "La suite, après la tabulation : \tJe suis loin."
        << std::endl;
    std::cout <<
        "Ces mots sont sur une ligne.\nEt ceux là sur la suivante.\n";

    return 0;
}
```

Retourner au texte.

Contenu masqué n°5

```cpp
#include <iostream>

int main()
{
    int nombre { 1 };
    int const multiplicateur { 2 };

    nombre = nombre * multiplicateur;
    std::cout << "Maintenant, mon nombre vaut " << nombre <<
        std::endl;

    nombre = nombre * multiplicateur;
    std::cout << "Maintenant, mon nombre vaut " << nombre <<
        std::endl;

    nombre = nombre * multiplicateur;
    std::cout << "Maintenant, mon nombre vaut " << nombre <<
        std::endl;

    return 0;
}
```

Vous remarquerez qu'on répète quand même plusieurs fois les mêmes instructions. Ce n'est pas grave. Nous verrons très bientôt comment remédier à ce problème. L'essentiel est que vous ayez pu voir l'avantage d'utiliser des variables.

Retourner au texte.

5. Le conditionnel conjugué en C++

« *Le conditionnel est un mode employé pour exprimer un état soumis à une condition.* » Telle est la définition qu'en donne Wikipédia . Cela vous rappelle-t-il vos cours de français ? Mais quel peut bien être le rapport avec la programmation ? C'est que **C++ permet aussi d'exprimer des conditions** et donc de modifier le comportement de notre programme.

Ce chapitre va donc introduire **les conditions** à notre palette d'outils et permettra de résoudre la problématique soulevée en conclusion du chapitre précédent.

5.1. Les booléens

Les conditions en C++ sont régies par un principe simple, qui est qu'**une condition est soit vraie, soit fausse**. Finalement, c'est un peu comme dans la réalité. Si je vous pose la question « Êtes-vous majeurs ? », la réponse est soit oui, soit non. De même, « Faîtes-vous 1m80 ou plus ? » entraînera soit une réponse positive, soit négative.

C++ nous offre un type conçu exprès pour ça, `bool`. Ce type peut prendre deux valeurs : soit `true`, signifiant vrai, soit `false` qui veut dire faux. Voyez par vous-mêmes l'exemple on ne peut plus bête ci-dessous.

```
int main()
{
    bool const vrai { true };
    bool const faux { false };

    return 0;
}
```

? D'accord, mais ça sert à quoi ça ? Je vois l'intérêt de manipuler des entiers ou du texte, mais stocker juste vrai ou faux, ça sert quand ?

L'intérêt semble en effet très faible si l'on se contente du code précédent. Sachez que les booléens sont partout, utilisables grâce à ce que l'on appelle **les opérateurs de comparaisons**. En voici la liste juste ci-dessous.

Opérateur	Signification	Exemple
`==`	**Égalité**, compare si deux variables sont égales entre elles.	`a == b`

`!=`	**Inégalité**, compare si deux variables sont de valeurs différentes.	`a != b`
`<`	**Strictement inférieur**, compare si la variable de gauche est strictement inférieure à celle de droite.	`a < b`
`<=`	**Inférieur ou égale**, compare si la variable de gauche est inférieure ou égale à celle de droite.	`a <= b`
`>`	**Strictement supérieur**, compare si la variable de gauche est strictement supérieure à celle de droite.	`a > b`
`>=`	**Supérieur ou égale**, compare si la variable de gauche est supérieure ou égale à celle de droite.	`a >= b`

> **!** Opérateur d'égalité
>
> Non, `==` n'est pas une erreur. Il faut bien deux signes `=`, pour différencier avec l'opération consistant à modifier la valeur d'une variable.

Le rapport avec les booléens ? Chacune des expressions suivantes renvoie `true` ou `false`. Faisons donc un petit test.

```
 1  #include <iostream>
 2
 3  int main()
 4  {
 5      int const a { 10 };
 6      int const b { 20 };
 7
 8      // Cette directive nous permet d'afficher true ou false. Par
            défaut, std::cout affiche 1 ou 0.
 9      std::cout << std::boolalpha;
10
11      std::cout << "a == b donne " << (a == b) << std::endl;
12      std::cout << "a != b donne " << (a != b) << std::endl;
13      std::cout << "a < b donne " << (a < b) << std::endl;
14      std::cout << "a <= b donne " << (a <= b) << std::endl;
15
16      // On peut tout à fait stocker le résultat dans une variable
            booléenne.
17      bool const plus_grand { a > b };
18      std::cout << "a > b donne " << plus_grand << std::endl;
19
20      bool const plus_grand_ou_egal { a >= b };
21      std::cout << "a >= b donne " << plus_grand_ou_egal <<
            std::endl;
22
23      return 0;
24  }
```

Lancez donc ce code, analysez les résultats. Sont-ils logiques ? Maintenant, changez la valeur des variables et observez de nouveau. Est-ce que c'est toujours logique ? Voyez-vous maintenant un peu plus l'utilité des booléens ?

5.2. if — Si, et seulement si…

Maintenant, examinons la façon de poser une question en C++, si l'on peut dire. Nous allons en effet découvrir la première structure de contrôle : `if`. Ce mot-clef est un mot anglais signifiant « *si* », et exécute des instructions si, et seulement si la condition donnée est vraie. Voyez donc ce schéma.

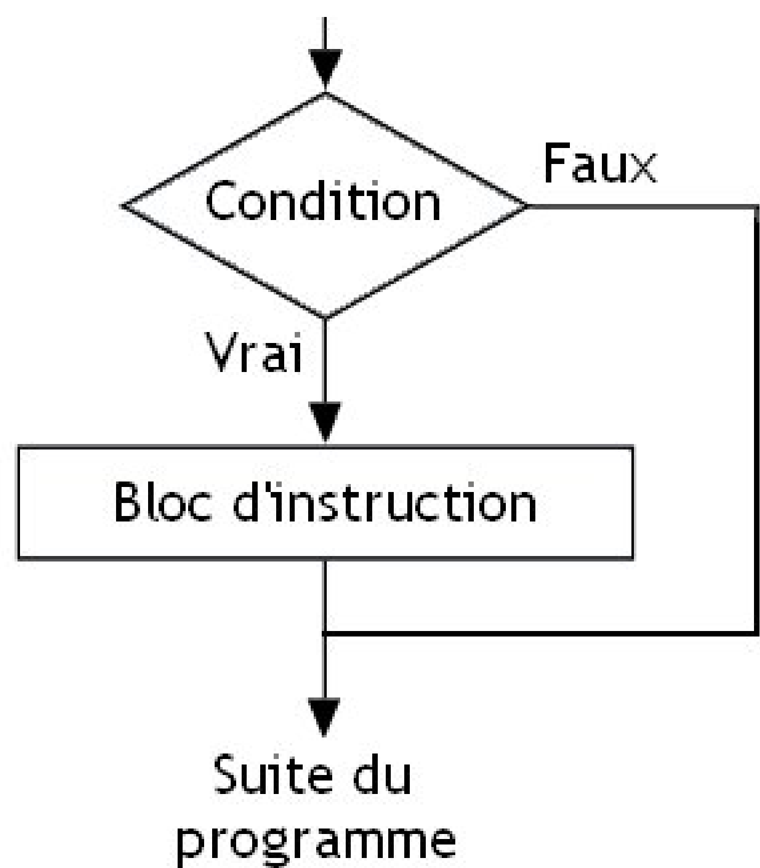


Figure 5.1. – Image originale tirée du tutoriel sur le langage C .

En C++, voici à quoi ressemble cette instruction. Toutes les instructions entre accolades seront exécutées si `condition` est vraie.

```
if (/* condition */)
{
    // Code à exécuter si la condition est vraie.
}
```

Prenons un exemple. Affichons la note à un examen, rentrée par l'utilisateur. Si celle-ci est supérieure ou égale à 16, nous afficherons un message de félicitations.

```
#include <iostream>

int main()
{
    std::cout << "Indique-moi ta note : ";

    int note { 0 };
    std::cin >> note;

    std::cout << "Tu as obtenu " << note << std::endl;

    if (note >= 16)
    {
        std::cout << "Félicitations, c'est une très bonne note !"
            << std::endl;
    }

    return 0;
}
```

Faites-le tourner. Vous verrez, le message de félicitations ne s'affiche que si vous rentrez un nombre supérieur ou égal à 16. Essayez avec 15, vous ne verrez plus le message.

5.2.1. À portée

Nous pouvons également **déclarer des variables** au sein du bloc d'accolades. Cependant, celles-ci ne sont utilisables que **jusqu'à l'accolade fermante correspondante**. Ainsi, le code ci-dessous n'est pas correct.

```
#include <iostream>

int main()
{
    if (true)
    {
        int const age { 42 };
        // D'autres instructions.

        // Aucun problème.
        if (age == 42)
        {
            // Ici non plus.
        }
```

```
15      }
16
17      // NON ! Le compilateur ne connait pas la variable age.
18      std::cout << age << std::endl;
19
20      return 0;
21  }
```

Ce sont les règles de C++ qui veulent ça. Quand on écrit un nouveau bloc à base d'accolades, on dit qu'on crée **une nouvelle portée**. Et une variable n'est utilisable que dans la portée, ou le bloc d'accolade, où elle a été déclarée.

C'est **une bonne pratique de déclarer ses variables dans la plus petite portée possible**. Il y a plusieurs raisons à ça.

— Dans le cas d'un code long et complexe, déclarer une variable au plus près possible de son utilisation permet de ne pas avoir à parcourir de longues lignes pour en obtenir des détails (quel est son type, sa valeur de base, etc). Cela aide à **la lecture et la compréhension du code**.
— Lorsqu'on atteint la fin du bloc correspondant, le programme **libère dans la mémoire les emplacements qu'il avait réservés** pour les variables qui s'y trouvaient. Dans le cas d'un `int` comme ici, ça ne change rien. Mais dans le cas de variables plus complexes que nous verrons plus tard, cela peut avoir une grande incidence sur les performances du programme.

Ainsi, je vous encourage fortement à déclarer vos variables à l'intérieur de vos blocs de condition **si elles ne sont pas destinées à être utilisées ailleurs**.

5.3. else — Sinon...

Maintenant que nous avons vu comment changer le code si une condition est vérifiée, voyons comment faire l'opposé, c'est-à-dire comment agir si celle-ci est fausse. Imaginons un programme demandant si l'utilisateur est majeur ou mineur. On peut imaginer quelque chose comme ce qui suit.

```
 1  #include <iostream>
 2
 3  int main()
 4  {
 5          std::cout << "Donne-moi ton âge: ";
 6          int age { 0 };
 7          std::cin >> age;
 8
 9          if (age >= 18)
10          {
11                  std::cout << "Tu es majeur." << std::endl;
```

```
12          }
13  
14          if (age < 18)
15          {
16                  std::cout << "Tu es mineur." << std::endl;
17          }
18  
19      return 0;
20  }
```

Ce programme est fonctionnel mais un peu lourd, puisqu'on répète deux fois une condition quasiment identique. Heureusement, C++ nous offre un autre mot-clef, `else`, qui exécute des instructions si la condition du `if` est fausse. Voyez-vous mêmes le code ci-dessous.

```
1   #include <iostream>
2  
3   int main()
4   {
5           std::cout << "Donne-moi ton âge: ";
6           int age { 0 };
7           std::cin >> age;
8  
9           if (age >= 18)
10          {
11                  std::cout << "Tu es majeur." << std::endl;
12          }
13          else
14          {
15                  std::cout << "Tu es mineur." << std::endl;
16          }
17  
18      return 0;
19  }
```

Le code se lit ainsi en français : « *Si l'âge rentré par l'utilisateur est supérieur ou égal à 18 ans, alors afficher "Tu es majeur", sinon afficher "Tu es mineur".* » Le mot-clef `else` est en effet un mot d'anglais qui existe et qui signifie, comme vous l'avez compris, « *sinon* ».

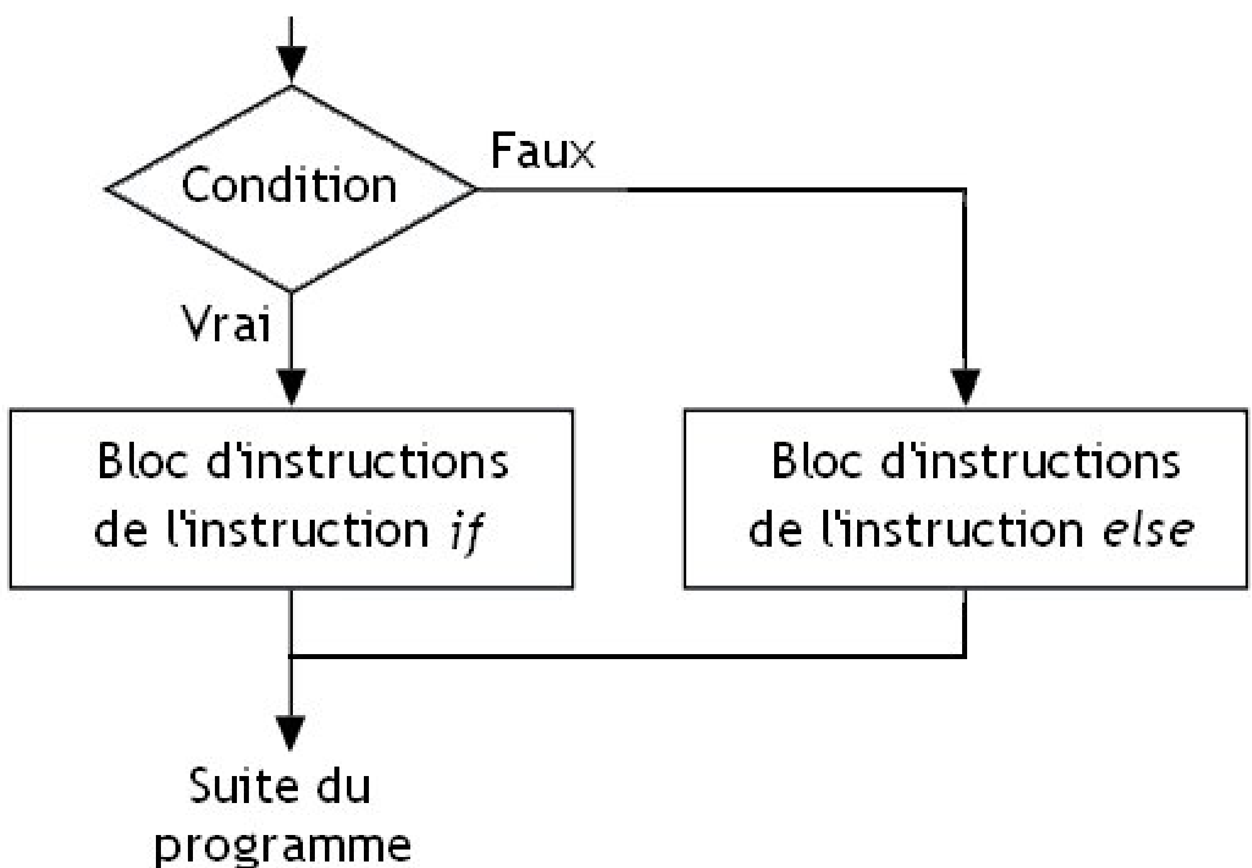


FIGURE 5.2. – Image originale tirée du [tutoriel sur le langage C](https ://zestedesavoir.com/contenus/755/le-langage-c-1/1042_les-bases-du-langage-c/4294_les-selections/#1-12882_la-structure-if).

Notez bien que, contrairement au `if`, `else` n'a pas de parenthèse pour préciser une condition. En effet, rappelons que `else` signifie « tout le reste ». Y'a t-il donc un moyen de tester plusieurs conditions différentes avant de faire « tout le reste » ? Sommes-nous condamnés à utiliser une suite de `if` et de `else` ?

5.4. else if — La combinaison des deux précédents

Comme vous l'avez vu en lisant le titre de cette section, C++ est un langage bien conçu qui répond à la problématique de la section précédente. La combinaison `else if` s'utilise en effet entre un `if` et un `else` pour dire « ou si cette condition est vraie. » Voyons déjà à quoi ressemble la syntaxe pour vous donner une idée.

```
1 if (/* condition 1 */)
2 {
3         // Si
4         // Code à exécuter si la condition 1 est vraie.
5 }
6 else if (/* condition 2 */)
```

```
{
        // Ou si.
        // Code à exécuter si la condition 1 est fausse mais que la
            condition 2 est vraie.
}
else if (/* condition 3 */)
{
        // Ou si.
        // Code à exécuter si ni la condition 1 ni la condition 2
            ne sont vraies mais que la condition 3 est vraie.
}
else
{
        // Sinon.
        // Code à exécuter si aucune condition n'est vraie.
}
```

Vous voyez le principe ? Bien, passons donc à un exercice concret. Imaginons un programme qui affiche des messages différents en fonction de l'âge de l'utilisateur. Mettez le texte et choisissez les âges que vous souhaitez. Une solution parmi d'autre se trouve ci-dessous.

Contenu masqué n°6

5.5. Conditions imbriquées

Il est tout à fait possible de **tester des conditions au sein d'autres conditions**. Si l'on prend l'exemple d'un menu de restaurant, on peut imaginer demander au client s'il veut de la viande ou un plat végétarien. Dans le premier cas, il faut lui demander la cuisson souhaitée ; dans le deuxième, on lui laisse le choix du légume. Cela est tout à fait possible en C++.

```
#include <iostream>

int main()
{
    std::cout << "Bienvenue au restaurant \"Chez Clem\"." <<
        std::endl;

    bool const plat_vegetarien { true };

    if (plat_vegetarien)
    {
        bool const salade { true };
        if (salade)
        {
```

```
14              std::cout << "Salade grecque ? De pâtes ?" <<
                    std::endl;
15          }
16          else
17          {
18              std::cout <<
                    "Nous avons également de la purée, une macédoine de légumes, ui
                    << std::endl;
19          }
20      }
21      else
22      {
23          bool const viande { true };
24          if (viande)
25          {
26              std::cout << "Bleue, saignante, à point, bien cuite ?"
                    << std::endl;
27          }
28          else
29          {
30              std::cout <<
                    "Sinon niveau poisson il y a pavé de saumon, filet de cabillaud
                    << std::endl;
31          }
32      }
33
34      return 0;
35  }
```

Instruction condensée

L'instruction `else if`, vue plus haut, correspond en fait à un `if` dans un `else`.

```
if (/* condition 1 */)
{
}
else if (/* condition 2 */)
{
}

// Équivaut à ceci.

if (/* condition 1 */)
{
}
else
{
    if (/* condition 2 */)
    {
    }
}
```

5.6. [T.P] Gérer les erreurs d'entrée — Partie I

Vous souvenez-vous de la problématique du chapitre précédent ? Nous avions des problèmes quand l'utilisateur rentrait du texte à la place d'un entier, par exemple. *Eh* bien **nous pouvons déjà apporter une première solution** à ce problème, solution que nous améliorerons au fur et à mesure.

Les développeurs ayant écrit la bibliothèque standard C++ ont déjà tout prévu. Trois fonctions, déjà programmées, vont nous intéresser.

— `std::cin > x` est en fait une fonction qui renvoie `true` si tout est correct ou `false` si on a rencontré une erreur lors de la saisie.
— `std::cin.clear()` restaure `std::cin` à un état fonctionnel, sans erreur.
— `std::cin.ignore()` permet d'ignorer un nombre défini de caractères, soit jusqu'à un nombre maximum (exemple `500`), soit jusqu'à un caractère précis (exemple `'\n'` ou `'a'`). Dans notre cas, nous allons utiliser ceci :

```
std::cin.ignore(255, '\n');
```

Maintenant, vous avez toutes les informations nécessaires pour commencer la protection de nos entrées. À vous de jouer !

Correction T.P Partie I

Voilà, nous avons déjà une petite protection. Mais elle ne marche qu'une fois : impossible de la faire marcher tant que l'utilisateur n'a pas rentré quelque chose de valide. Pour l'instant, laissez ça de côté, nous en reparlerons dans le prochain chapitre.

5.7. La logique booléenne

Nous avons vu les différents moyens de tester des conditions, ainsi que différents opérateurs de comparaisons qui retournent des booléens. Mais ce ne sont pas les seuls. Cette section va introduire l'algèbre booléenne, et rassurez-vous, c'est simple à comprendre, puissant et bien utile.

5.7.1. AND — Tester si deux conditions sont vraies

Imaginons un instant que vous souhaitiez faire l'acquisition d'une voiture. Pour cela, il vous faut remplir deux conditions : avoir le permis et suffisamment d'argent. Si l'une des deux conditions n'est pas remplie, ou les deux, vous continuerez à prendre le bus. Le seul moyen de faire hurler le moteur sur l'autoroute, c'est d'avoir assez d'argent **ET** d'avoir le permis.

C++ nous fournit deux moyens d'exprimer cet opérateur logique « ET » : `&&` et `and`. Les deux sont parfaitement valables et interchangeables. Le deuxième était notamment prévu à l'époque où tous les claviers ne possédaient pas le symbole &.

> **!** Ancienne spécificité Microsoft
>
> Si `and` ne marche pas avec Visual Studio, il faut inclure le fichier `<ciso646>`. Avec une version à jour, ce n'est cependant plus nécessaire.

```
1  #include <iostream>
2
3  int main()
4  {
5      int const prix_voiture { 5000 };
6      int const argent { 2000 };
7      bool const a_le_permis { true };
8
9      if (argent >= prix_voiture && a_le_permis)
10     {
11         std::cout << "Voici les clés, bonne route." << std::endl;
12     }
13     else
14     {
15         std::cout << "Désolé, vous allez devoir prendre le bus." <<
               std::endl;
16     }
17
```

```
18        return 0;
19    }
```

Testez donc ce programme en modifiant le prix de la voiture, l'argent disponible ou la possession ou non du permis. Comme écrit plus haut, il n'y a que si `a_le_permis` est à `true` et qu'il y a plus d'argent que le prix de la voiture que vous serez en mesure d'avoir les clés.

Voici ce qu'on appelle la table de vérité de l'opérateur AND, qui formalise les entrées et les sorties de cet opérateur. N'hésitez pas à en lire plus sur le sujet s'il vous intéresse.

Première condition	Seconde condition	Résultat de l'opérateur « AND »
false	false	false
false	true	false
true	false	false
true	true	true

5.7.2. OR — Tester si au moins une condition est vraie

Maintenant, nous allons aller visiter un musée. Quand on arrive devant l'entrée, bonne surprise, l'entrée est gratuite si l'on a moins de 26 ans **OU** si l'on est professeur (peu importe la matière). Il suffit de remplir l'une des deux conditions pour être admis gratuitement à l'intérieur. Bien sûr, si vous remplissez les deux, l'offre reste valable.

Cet exemple va servir de base pour l'introduction de l'opérateur C++ « OU » : au choix, `||` (qui se tape sur un AZERTY français en faisant Alt Gr + - (le numéro 6) ou `or`, présent pour la même raison que précédemment.

! Ancienne spécificité Microsoft

Si `or` ne marche pas avec Visual Studio, il faut inclure le fichier `<ciso646>`. Avec une version à jour, ce n'est cependant plus nécessaire.

```
1   #include <iostream>
2
3   int main()
4   {
5       int const age { 25 };
6       bool const est_professeur { true };
7
8       if (age <= 26 || est_professeur)
9       {
10          std::cout << "Bonne visite." << std::endl;
11      }
```

```
12      else
13      {
14          std::cout << "Désolé, il va falloir payer." << std::endl;
15      }
16
17      return 0;
18  }
```

Encore une fois, faites vôtre ce programme et modifiez-le. Voyez dans quelles circonstances l'entrée est gratuite et à quels moments il faut payer. Vous allez obtenir des résultats identiques à la table de vérité ci-dessous.

Première condition	Seconde condition	Résultat de l'opérateur « OR »
false	false	false
false	true	true
true	false	true
true	true	true

5.7.3. NOT — Tester la négation

Il fait chaud, vous allez à la piscine vous rafraichir. En arrivant, vous jetez un œil au règlement et vous voyez qu'il est interdit de nager si l'on a un short de bain et non un maillot de bain. Vous raisonnez donc ainsi : « *Si j'ai un maillot de bain, alors tout va bien, l'entrée m'est autorisée, je peux nager. Si j'ai un short de bain, alors l'entrée m'est interdite et je dois en acheter un avant de nager.* » Comment exprimeriez-vous ça en C++ ? Peut-être avec le code ci-dessous ?

```
1   int main()
2   {
3       bool const ai_je_un_maillot { true };
4       if (ai_je_un_maillot)
5       {
6           // Je n'ai rien à faire, tout est OK.
7       }
8       else
9       {
10          // Je dois aller acheter un maillot.
11      }
12
13      // Je nage et je suis heureux.
14      return 0;
15  }
```

Ce code fonctionne, mais il nous pose un problème : on a un `if` vide et c'est moche. L'idéal

serait de ne traiter que le cas « Pas de maillot de bain ». C++ nous permet justement cela avec l'opérateur de négation `!` ou `not`. Celui-ci renvoie la valeur inverse de l'expression testée. Si l'expression vaut `true`, alors cet opérateur renverra `false` ; si, au contraire, elle valait `false`, sa négation donnera `true`.

> **!** Ancienne spécificité Microsoft
>
> Si `not` ne marche pas avec Visual Studio, il faut inclure le fichier `<ciso646>`. Avec une version à jour, ce n'est cependant plus nécessaire.

```cpp
int main()
{
    bool const ai_je_un_maillot { true };
    if (!ai_je_un_maillot)
    {
        // Je dois aller acheter un maillot.
    }

    // Je nage et je suis heureux.
    return 0;
}
```

Notre code devient plus court, plus concis et plus agréable à lire, il gagne en qualité. Je vous mets en dessous la table de vérité, bien plus courte cette fois.

Valeur de la condition	Résultat de l'opérateur « NOT »
true	false
false	true

> **?** Il y a juste quelque chose que je ne comprends pas. Pourquoi ne pas créer tout simplement un booléen inverse, genre `pas_de_maillot` ? Pourquoi se casser la tête à tester la négation d'une condition et ne pas tester la condition inverse tout de suite ?

C'est une très bonne question. Dans un exemple aussi simpliste que celui-ci, monté de toute pièce, on aurait pu, c'est vrai. Mais d'autres fois nous n'avons pas le choix, car nous utilisons du code qui a été programmé d'une certaine façon.

5.8. Tester plusieurs expressions

Nous ne sommes absolument pas limités à tester deux expressions en même temps. On peut le faire avec trois, quatre, dix, soixante (bien que soixante soit un exemple extrême et vraiment trop peu lisible). Le code suivant le prouve.

```cpp
#include <iostream>

int main()
{
    bool const ai_je_envie_de_nager{ true };
    bool const ai_je_un_maillot { true };
    int const argent { 20 };
    int const prix { 5 };

    if (ai_je_envie_de_nager && ai_je_un_maillot && argent >= prix)
    {
        std::cout << "PLOUF !" << std::endl;
    }
    else
    {
        std::cout << "Huum, un autre jour." << std::endl;
    }

    return 0;
}
```

Dans le cas du même opérateur, ils sont évalués de gauche à droite. C'est le cas du code précédent. Mais que se passe t-il **si l'on mélange plusieurs opérateurs différents** dans la même expression ? Sauriez-vous dire ce que quelque chose comme `expression1 && expression2 || expression3` va donner ? En effet, les opérateurs logiques sont comme les opérateurs mathématiques que nous avons vus dans les chapitres précédents : **ils ont une priorité**.

1. Le plus prioritaire, c'est la négation `!`.
2. Ensuite vient le « ET » `&&`.
3. Enfin, le « OU » `||` est le moins prioritaire.

Ainsi, dans l'exemple `!a && b`, c'est d'abord `!a` qui est évalué, puis la nouvelle valeur et `b` sont testées avec l'opérateur ET `&&`. Avec le code `a && b || c && d`, dans l'ordre, on évalue `a && b`, `c && d` et enfin `a && b || c && d`.

Bien que la priorité des opérateurs soit clairement définie par C++, une bonne pratique consiste à **ajouter des parenthèses** autour des expressions pour rendre le **code plus lisible et plus clair**. Si l'on reprend nos exemples, cela donne `(!a) && b` pour le premier et `(a && b) || (c && d)` pour le deuxième. C'est clair ? Prouvez-le-moi en mettant les parenthèses au bon endroit dans ces codes (piqués à @gbdivers).

```cpp
a && b && c
a || b || c
a || b && c
a && b || c

```

```
 6 !a && b
 7 a || !b
 8
 9 a && b || c && d
10 a || b && c || d
```

Correction

Parenthèses

À partir de maintenant, je vais mettre des parenthèses dans le code pour le rendre clair et explicite.

5.8.1. Évaluation en court-circuit

Imaginez un formulaire à remplir, avec de multiples questions, pour demander une place de parking dans votre immeuble. Pour ça, il faut déjà habiter l'immeuble en question et avoir une voiture. Si vous habitez ailleurs, pas la peine de réfléchir à la possession ou non d'une voiture, vous êtes inéligible pour cette place de parking.

Eh bien le compilateur mène le même raisonnement quand il évalue une expression constituée de « AND ». S'il détecte qu'une condition est fausse, alors **il ne sert à rien d'évaluer le reste de l'expression** puisque le résultat est forcément faux. Si vous ne voyez pas pourquoi, retournez un peu plus haut jeter un œil à la table de vérité de « AND ».

```
 1 #include <iostream>
 2
 3 int main()
 4 {
 5     bool const ai_je_une_voiture { false };
 6     bool const habite_immeuble { true };
 7
 8     // Comme ai_je_une_voiture vaut false, le compilateur ne va pas
           tester l'expression habite_immeuble.
 9     if (ai_je_une_voiture && habite_immeuble)
10     {
11         std::cout << "Voici votre place de parking." << std::endl;
12     }
13     else
14     {
15         std::cout << "Désolé, vous n'êtes pas éligible." <<
               std::endl;
16     }
17
```

```
18      return 0;
19  }
```

Avec l'opérateur « OR », le même principe s'applique : si l'une des expressions est évaluée à `true`, alors, selon la table de vérité de l'opérateur « OR », le résultat sera `true`, donc pas la peine d'évaluer le reste.

Ce principe d'optimisation par le compilateur s'appelle **l'évaluation en court-circuit**. Il permet d'optimiser l'exécution du code en ne perdant pas de temps sur des instructions qui seront forcément évaluées à `false` ou à `true`. Quand nous avancerons dans notre apprentissage du C++, nous verrons des cas où l'évaluation en court-circuit est bien utile.

5.9. Exercices

Allez, c'est l'heure de pratiquer un peu. C'est le meilleur moyen de progresser et de fixer toutes ces nouvelles notions dans votre mémoire. N'hésitez pas à faire cette section à tête reposée et avec le cours sous les yeux en même temps.

> **Un mot sur la licence**
>
> Certains exercices sont adaptés du cours sur le langage C, comme m'y autorisent la licence et mon statut de double auteur.

5.9.1. Une pseudo-horloge

Demandons à l'utilisateur de rentrer un nombre et nous lui afficherons la période de la journée correspondante : nuit, matin, après-midi, soir. Ainsi, entre 8h et 12h, nous sommes le matin. Mais attention, nous voulons aussi gérer les cas un peu spéciaux que sont midi, minuit et un nombre qui n'est pas entre 0 et 24.

Correction pseudo-horloge

5.9.2. Score

Imaginez que vous avez un score de jeu vidéo sous la main.

— Si le score est strictement inférieur à deux mille, affichez « C'est la catastrophe!»
— Si le score est supérieur ou égal à deux mille et que le score est strictement inférieur à cinq mille, affichez: « Tu peux mieux faire!»
— Si le score est supérieur ou égal à cinq mille et que le score est strictement inférieur à neuf mille, affichez: « Tu es sur la bonne voie!»
— Sinon, affichez: « Tu es le meilleur!»

Correction score

5.9.3. XOR — Le OU exclusif

Le OR que nous avons vu plus tôt dans ce chapitre est dit « inclusif ». Ainsi, `a || b` signifie `a`, ou `b`, ou les deux. Le but de cet exercice va être de faire un OU dit « exclusif », c'est-à-dire que si `a` et `b` renvoient la même chose, l'expression est évaluée à `false`, sinon à `true`. Vous pouvez le faire avec les opérateurs que nous avons vus.

Indice

Correction XOR

5.9.4. En résumé

— Nous pouvons tester une expression avec `if`. Si celle-ci est fausse, on peut tester d'autres expressions avec autant de `else if` qu'on veut. Enfin, `else` permet de tester le cas général, « tout le reste ».
— Nous pouvons comparer des expressions entre elles avec les opérateurs de comparaisons.
— Les opérateurs logiques permettent de tester plusieurs expressions en même temps ou d'en obtenir la négation.
— Le compilateur optimise l'évaluation de ces multiples expressions grâce à l'évaluation en court-circuit.

Contenu masqué

Contenu masqué n°6

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Quel âge as-tu ? ";
6
7      unsigned int age { 0 };
8      std::cin >> age;
9
```

```
10      if (age < 5u)
11      {
12          std::cout <<
                "Tu n'es pas un peu jeune pour naviguer sur internet ?"
                << std::endl;
13      }
14      else if (age < 18u)
15      {
16          std::cout << "Tu es mineur !" << std::endl;
17      }
18      else if (age < 30u)
19      {
20          std::cout << "Vous êtes majeur !" << std::endl;
21      }
22      else if (age < 50u)
23      {
24          std::cout <<
                "Vous n'êtes plus très jeunes, mais vous n'êtes pas encore vieux."
                << std::endl;
25      }
26      else if (age < 70u)
27      {
28          std::cout << "Vous commencez à prendre de l'âge." <<
                std::endl;
29      }
30      else if (age < 120u)
31      {
32          std::cout <<
                "Vous avez du en voir, des choses, durant votre vie !"
                << std::endl;
33      }
34      else
35      {
36          std::cout << "Quelle longévité !" << std::endl;
37      }
38
39      return 0;
40  }
```

Retourner au texte.

Contenu masqué n°7 :

Correction T.P Partie I

```cpp
#include <iostream>
#include <string>

int main()
{
    std::cout << "Entre ton age : ";
    unsigned int age { 0 };

    if (std::cin >> age)
    {
        // Tout va bien.
        std::cout << "Tu as " << age << " ans.\n";
    }
    else
    {
        // Si std::cin a rencontré une erreur quelconque.
        std::cout <<
            "Tu n'as pas rentré un entier, il y a eu une erreur."
            << std::endl;
        std::cin.clear(); // On remet std::cin dans un état
            fonctionnel.
        std::cin.ignore(255, '\n'); // On vide les caractères
            mémorisés.
    }

    std::cout << "Maintenant, vérifions que tout va bien." <<
        std::endl;
    std::cout << "Entre ton nom : ";
    std::string nom { "" };
    std::cin >> nom;
    std::cout << "Tu t'appelles " << nom << ".\n";

    return 0;
}
```

Si vous avez eu du mal ou n'avez pas trouvé, ce n'est pas grave, vous êtes en train d'apprendre. N'hésitez pas à relire la solution jusqu'à ce que tout vous paraisse limpide. Retourner au texte.

Contenu masqué n°8 : Correction

Voici la solution. L'avez-vous trouvé ? Vous êtes sur la bonne voie ! Vous avez-eu du mal ? N'hésitez pas à relire les passages précédents à tête reposée et à faire vos propres exemples.

```
(a && b) && c
(a || b) || c
a || (b && c)
(a && b) || c

(!a) && b
a || (!b)

(a && b) || (c && d)
(a || (b && c)) || d
```

Retourner au texte.

Contenu masqué n°9 : Correction pseudo-horloge

```cpp
#include <iostream>

int main()
{
    std::cout << "Quelle heure est-il ?" << std::endl;
    int heure { 0 };
    std::cin >> heure;

    if (heure > 0 && heure < 7)
    {
        std::cout << "Zzz..." << std::endl;
    }
    else if (heure >= 7 && heure < 12)
    {
        std::cout << "C'est le matin !" << std::endl;
    }
    else if (heure == 12)
    {
        std::cout << "Il est midi !" << std::endl;
    }
    else if (heure > 12 && heure < 18)
    {
        std::cout << "C'est l'après-midi !" << std::endl;
    }
    else if (heure >= 18 && heure < 24)
    {
        std::cout << "C'est le soir !" << std::endl;
    }
    else if (heure == 24 || heure == 0)
```

```cpp
    {
        std::cout << "Il est minuit, dormez brave gens !" <<
            std::endl;
    }
    else
    {
        std::cout <<
            "Il est l'heure de réapprendre à lire l'heure !" <<
            std::endl;
    }

    return 0;
}
```

Retourner au texte.

Contenu masqué n°10 : Correction score

```cpp
#include <iostream>

int main()
{
    std::cout << "Quel est le score du joueur ? " << std::endl;
    int score { 0 };
    std::cin >> score;

    if (score < 2000)
    {
        std::cout << "C'est la catastrophe !" << std::endl;
    }
    else if (score >= 2000 && score < 5000)
    {
        std::cout << "Tu peux mieux faire !" << std::endl;
    }
    else if (score >= 5000 && score < 9000)
    {
        std::cout << "Tu es sur la bonne voie !" << std::endl;
    }
    else
    {
        std::cout << "Tu es le meilleur !" << std::endl;
    }

    return 0;
}
```

Retourner au texte.

Contenu masqué n°11 : Indice

Si vous bloquez, voici déjà la table de vérité de l'opérateur.

Première condition	Seconde condition	Résultat de l'opérateur « XOR »
false	false	false
false	true	true
true	false	true
true	true	false

Retourner au texte.

Contenu masqué n°12 : Correction XOR

La solution s'écrit ainsi : `(a && !b) || (b && !a)`. Vous pouvez vérifier avec le programme ci-dessous.

```
#include <iostream>

int main()
{
    std::cout << std::boolalpha;

    bool const deux_false { (false && !false) || (false && !false)
        };
    std::cout << "(false && !false) || (false && !false) == " <<
        deux_false << std::endl;

    bool const true_false { (true && !false) || (false && !true) };
    std::cout << "(true && !false) || (false && !true) == " <<
        true_false << std::endl;

    bool const false_true { (false && !true) || (true && !false) };
    std::cout << "(false && !true) || (true && !false) == " <<
        false_true << std::endl;

    bool const deux_true { (true && !true) || (true && !true) };
    std::cout << "(true && !true) || (true && !true) == " <<
        deux_true << std::endl;
```

```

    return 0;
}
```

Sinon vous pouvez utiliser `xor`, si vous incluez bien `<ciso646>` avec Visual Studio. Retourner au texte.

6. Des boucles qui se répètent, répètent, répètent…

Nous sommes maintenant capables d’exécuter des codes différents en fonction de conditions. Mais notre programme reste très linéaire : nous exécutons les instructions l’une après l’autre, du début à la fin, de haut en bas. Dans la droite ligne du chapitre précédent, nous allons donc voir un mécanisme offert par C++ pour **répéter autant de fois que l’on souhaite une série d’instructions**.

Ce chapitre introduira **les boucles** et permettra d’améliorer encore notre gestion des erreurs, vue dans le chapitre précédent.

6.1. while — Tant que…

Commençons ce chapitre avec une boucle `while`. En toute originalité, ce mot-clef est aussi un mot anglais qui signifie « *tant que* ». Elle exécute une série d’instructions **tant que la condition est vraie**. Par contre, dès que la condition est évaluée à `false`, la boucle s’arrête. On dit aussi qu’**on boucle** ou qu’**on itère**.

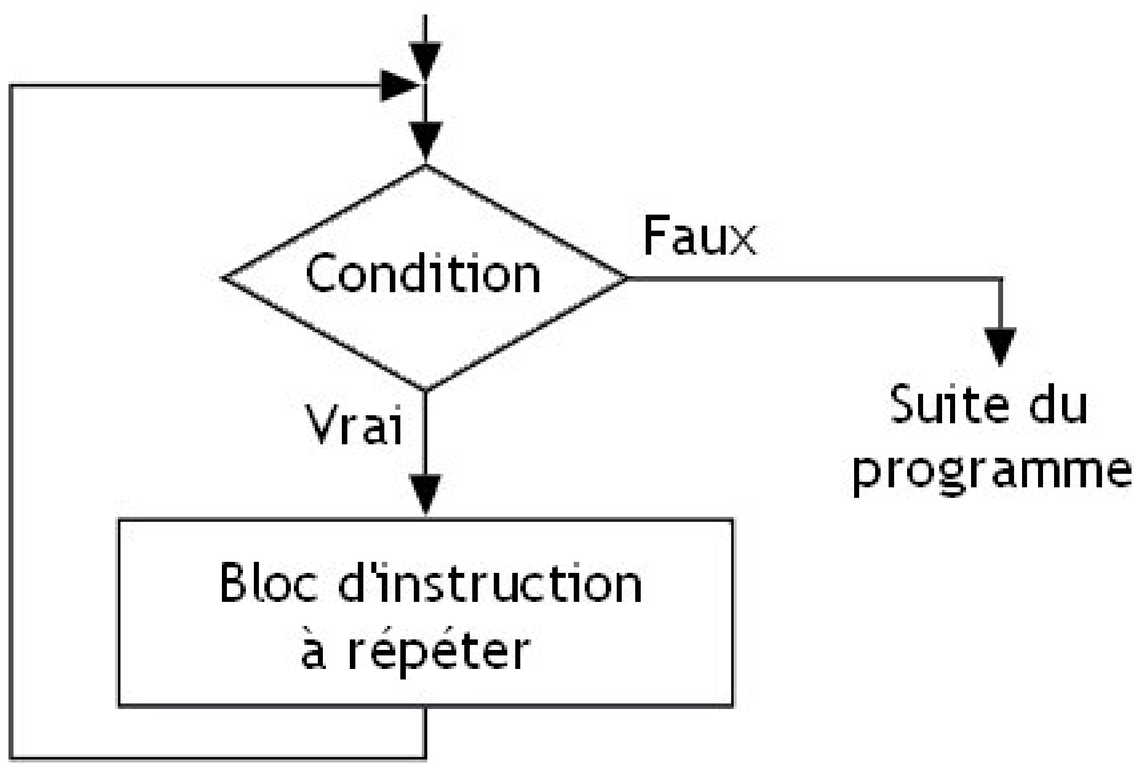


Figure 6.1. – Image originale tirée du tutoriel sur le langage C .

Prenons un exemple tout bête : notre programme va compter de 0 à 9. Voici le code correspondant.

```
#include <iostream>

int main()
{
    std::cout << "Salut, je vais compter de 0 à 9." << std::endl;
    int compteur { 0 };

    while (compteur < 10)
    {
        std::cout << compteur << std::endl;
        ++compteur;
    }

    std::cout << "Je m'arrête, après je sais pas faire." <<
        std::endl;
    return 0;
}
```

Le code est exécuté ainsi : on évalue la condition. Tant que `compteur` est strictement inférieur à 10, alors on affiche le nombre puis on l'incrémente. Quand finalement `compteur` vaut 10, la condition devient fausse, on passe à la suite du code.

Boucle infinie

Il est important de vérifier qu'à un moment ou un autre la condition devienne fausse et qu'on sort bien de la boucle. Si l'on n'y prend pas garde, on tombe dans le piège de la **boucle infinie** !

```cpp
#include <iostream>

int main()
{
    std::cout << "Salut, je vais compter de 0 à 9." <<
        std::endl;
    int compteur { 0 };

    while (compteur < 10)
    {
        std::cout << compteur << std::endl;
        // Oups, on a oublié d'incrémenter le compteur. Nous
            sommes bloqués dans la boucle.
    }

    std::cout << "Je ne serai jamais affiché." << std::endl;
    return 0;
}
```

Dans ces cas, il ne reste plus qu'à terminer le programme sauvagement en le tuant (oui les informaticiens sont brutaux). Je répète donc, **faites attention aux boucles infinies**.

Comme les boucles sont basées sur les conditions, aucun problème pour en utiliser plusieurs en même temps.

```cpp
#include <iostream>

int main()
{
    char entree { '?' };
    int compteur { 0 };

    // On boucle tant qu'on a pas rentré la lettre 'e' ou que le
        compteur n'a pas atteint 5.
    while (compteur < 5 && entree != 'e')
    {
        std::cout << "Le compteur vaut " << compteur << std::endl;
        std::cout << "Rentre un caractère : ";
        std::cin >> entree;

        ++compteur;
    }

    std::cout << "Fin du programme." << std::endl;
    return 0;
}
```

6.2. Exercices

6.2.1. Une laverie

Nous allons faire un **programme d'aide dans une laverie automatique**. Dans celle-ci, deux types de machines sont employées : des machines économes capables de laver 5kg de linge au maximum et d'autres, plus gourmandes, capables de laver 10kg.

- Si le linge fait moins de 5 kg, il faut le mettre dans une machine de 5kg.
- S'il fait entre 5kg et 10kg, il faut le mettre dans une machine de 10kg.
- Enfin, s'il fait plus, il faut répartir le linge entre machines de 10kg et, dès qu'il reste 5kg ou moins, mettre le linge restant dans une machine 5kg.

Correction laverie

6.2.2. PGCD

Allez, on va faire un peu de mathématiques ! Le PGCD de deux nombres est un entier qui est le plus grand diviseur que ces deux nombres ont en commun. Ainsi, le PGCD de 427 et 84 est 7, car $427 = 7 \times 61$ et $84 = 7 \times 12$. Pour le calculer, voici l'algorithme.

- On stocke le modulo du premier nombre, appelé `a`, par l'autre nombre, appelé `b`, dans une variable `r`.
- Tant que `r` est différent de zéro, on effectue les opérations suivantes.
  - On affecte la valeur de `b` à `a`.
  - On affecte la valeur de `r` à `b`.
  - On affecte à `r` le résultat du modulo de `a` par `b`.
- Quand `r` est nul, alors `b` représente le PGCD des deux nombres donnés en entrée.

Correction PGCD

6.2.3. Somme de nombres de 1 à n

Disons que je vous donne un entier N au hasard. Pouvez-vous me créer un programme capable de me donner la somme de tous les entiers de 1 à n, c'est-à-dire la somme de $1 + 2 + 3 + ... + (n - 2) + (n - 1) + n$?

Correction somme de 1 à n

6.3. do while — Répéter … tant que

La deuxième boucle offerte par C++ est très proche de la précédente. En fait, c'est la même, à deux différences près. Une boucle `do while` **s'exécute toujours une fois au minimum**, même si la condition est fausse. La condition est en effet vérifiée **après** l'exécution du code.

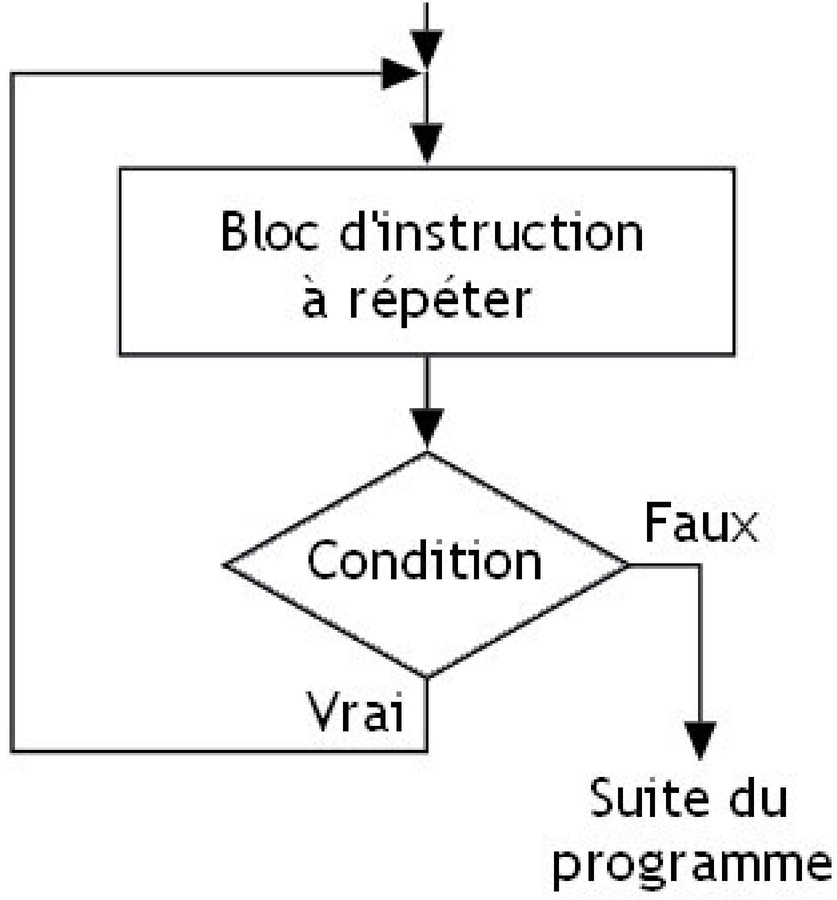


FIGURE 6.2. – Do while

La deuxième, c'est qu'il faut un point-virgule après le `while`. Voyez par vous-mêmes le code suivant.

```
#include <iostream>

int main()
{
    do
    {
        std::cout << "On passe quand même ici." << std::endl;
    } while (false);

    return 0;
}
```

Hormis ces deux subtiles différences, `do while` se comporte exactement comme `while` et permet elle aussi de boucler. Nous allons voir dans la section suivante comment les deux nous aiderons à résoudre le problème de sécurisation des entrées, soulevé au chapitre précédent.

6.4. [T.P] Gérer les erreurs d'entrée — Partie II

Dans la première partie, nous avions un mécanisme qui ne nous protégeait qu'une seule fois des erreurs. Maintenant, nous sommes en mesure de **nous protéger tant que l'entrée n'est pas correcte**. Améliorez donc le code que vous avez écrit pour qu'il soit capable de protéger le code suivant.

```cpp
#include <iostream>

int main()
{
    std::cout << "Quel jour es-tu né ? ";
    int jour { 0 };
    std::cin >> jour;

    std::cout << "Quel mois es-tu né ? ";
    int mois { 0 };
    std::cin >> mois;

    std::cout << "Tu es né le " << jour << "/" << mois << "." <<
        std::endl;
    return 0;
}
```

Correction T.P partie II

Petit à petit, grâce à nos connaissances qui s'étoffent, nous améliorons la protection des entrées. Mais c'est dommage d'avoir le même code dupliqué à plusieurs endroits. C'est sale, ça nous fait perdre du temps si l'on doit modifier un morceau de code dans chaque endroit ; et je ne parle pas des erreurs liés à des morceaux de codes différents des autres. Nous apprendrons dans le chapitre suivant comment s'en sortir.

6.5. for — Une boucle condensée

Les boucles `while` et `do while` sont très utiles et ajoutent une corde à notre arc. Mais ne trouvez-vous pas ça dommage d'avoir nos variables liées à l'itération (comme `compteur`) séparées de la boucle elle-même ?

Actuellement, notre code est simple, mais imaginons un instant un code bien plus complexe. Il serait mieux de **séparer les opérations liées à la boucle de nos autres variables**

importantes. Cela rendra le code plus clair et plus compréhensible. C'est exactement ce que permet `for` (« pour » en anglais).

```
for (/* initialisation */; /* condition */; /* itération */)
{
    // Code de la boucle.
}

// Est équivalent à la boucle ci-dessous.

// Notez la création d'une portée artificielle.
{
    // Initialisation.
    while (/* condition */)
    {
        // Code de la boucle.
        /* itération */
    }
}
```

Hormis cette syntaxe différente, `for` s'utilise comme ses consœurs. Nous sommes libres de l'initialisation, de la condition et de l'itération.

```
#include <iostream>

int main()
{
    // Compte de 10 à 1. On décrémente ici.
    for (int compteur {10}; compteur > 0; --compteur)
    {
        std::cout << compteur << std::endl;
    }

    std::cout << "Fin du programme." << std::endl;
    return 0;
}
```

Nous pouvons même effectuer plusieurs opérations en même temps, tant que les variables déclarées sont du même type. Cette possibilité est cependant à prendre avec des gants parce qu'**elle peut rendre le code très peu lisible**.

```
#include <iostream>

int main()
{
```

```
5       // Compte de 10 à 1 et de 1 à 10 en même temps.
6       for (int compteur {10}, compteur_inverse {1}; compteur > 0 &&
            compteur_inverse <= 10; --compteur, ++compteur_inverse)
7       {
8           std::cout << compteur << " et " << compteur_inverse <<
                std::endl;
9       }
10
11      std::cout << "Fin du programme." << std::endl;
12      return 0;
13  }
```

?

À quel moment dois-je utiliser `for` ou `while` ?

Le choix est vôtre, C++ vous laisse entièrement libre. Après, on utilise plus souvent `for` dans le cas où l'on veut **itérer** (compter de 1 à 100 par exemple), alors que `while` **boucle pour réaliser un traitement autant de fois que nécessaire** (par exemple, tant que `std::cin.fail` renvoie `true`).

i

Variante

Dans le chapitre suivant, nous verrons qu'il existe une autre boucle `for`.

6.6. Boucles imbriquées

Comme pour les conditions, il est tout à fait possible d'**imbriquer plusieurs boucles** l'une dans l'autre. De même, on peut tout à fait mélanger les types de boucles, comme le prouve l'exemple suivant, qui affiche les tables de multiplications de 1 à 10 en utilisant `while` et `for`.

```
1   #include <iostream>
2
3   int main()
4   {
5       int facteur_gauche { 1 };
6       while (facteur_gauche <= 10)
7       {
8           for (int facteur_droite { 1 }; facteur_droite <= 10;
                ++facteur_droite)
9           {
10              std::cout << facteur_gauche << "x" << facteur_droite <<
                    " = " << facteur_gauche * facteur_droite <<
                    std::endl;
```

```
11          }
12          // On saute une ligne pour séparer chaque table.
13          std::cout << std::endl;
14          ++facteur_gauche;
15      }
16
17      return 0;
18  }
```

6.7. Convention de nommage

Dernier point avant de continuer : vous verrez très souvent, tant dans ce tutoriel que dans les divers codes C++ que vous aurez l'occasion d'examiner, que les programmeurs utilisent `i` comme identifiant de variable pour parcourir une boucle (ainsi que `j` et `k` dans le cas de boucles imbriquées). C'est une abréviation pour *iterator*, parce que cette variable itère, répète le corps de la boucle. **Ce nom est tellement court et explicite qu'il est pour ainsi dire devenu une convention de nommage en C++**.

6.8. Contrôler plus finement l'exécution de la boucle

Les boucles sont très utiles, mais le code à l'intérieur des accolades est soit entièrement sauté, soit entièrement exécuté. Sachez que C++ permet de contrôler un peu plus précisément l'exécution avec deux mots-clefs : `break` et `continue`.

> **Précision**
>
> Simple précision : `break` et `continue` s'utilisent avec les boucles uniquement. Ils ne s'utilisent pas avec les conditions : pas de `break` au sein d'un `if`, par exemple.

6.8.1. `break` — Je m'arrête là

Le premier, c'est `break`, qui en anglais signifie littéralement « casser », « interrompre ». Celui-ci nous autorise à **mettre fin à l'exécution de la boucle**, peu importe où nous en étions. Dans le code ci-dessous, vous ne verrez jamais afficher les nombres au-dessus de 3 parce que la boucle est terminée par `break` dès que notre variable vaut 3.

```
1  #include <iostream>
2
3  int main()
4  {
```

```
5        for (int i { 0 }; i < 10; ++i)
6        {
7            // Si i vaut 3, j'arrête la boucle.
8            if (i == 3)
9            {
10               break;
11           }
12
13           std::cout << i << std::endl;
14       }
15
16       return 0;
17   }
```

Dans le cas de boucles imbriquées, `break` ne stoppe l'exécution que de la boucle dans laquelle il se trouve, mais pas les boucles englobantes.

```
1   for (int i { 0 }; i < 5; ++i)
2   {
3       for (int j { 0 }; j < 3; ++j)
4       {
5           if (j == 1)
6           {
7               // Quand j vaut 1, on retourne dans le for(i).
8               break;
9           }
10
11          std::cout << "Voici la valeur de j : " << j << std::endl;
12      }
13
14      std::cout << "Voici la valeur de i : " << i << std::endl;
15  }
```

Pour illustrer son fonctionnement avec un cas plus utile, prenons un exemple mathématique. Connaissez-vous le concept du test de primalité ? Il s'agit d'un test pour déterminer si un entier est premier, c'est-à-dire divisible uniquement par 1 et par lui-même ; 13 est un nombre premier mais pas 10, car divisible par 1, 2, 5, et 10. Essayez donc de faire cet exercice.

Indice

Correction

6.8.2. `continue` — Saute ton tour !

L'autre mot-clef, `continue`, permet de **sauter l'itération courante**. Toutes les instructions qui se trouvent après ce mot-clef sont donc ignorées et **la boucle continue au tour suivant**. Imaginons que nous voulions un programme permettant d'afficher les nombres impairs compris entre 0 et un nombre choisi par l'utilisateur. Essayez donc de le faire, avec les connaissances que vous avez.

Indice

Correction

Potentielle boucle infinie

Attention lorsque vous utilisez `continue` au sein d'une boucle `while` ou `do while`, parce que **TOUT le code situé après est sauté**, même les instructions d'incrémentation. Le code suivant est donc un exemple de boucle infinie.

```
int i { 0 };
while (i < 10)
{
    if (i == 5)
    {
        // Ohoh...
        continue;
    }

    ++i;
}
```

6.8.3. En résumé

- C++ nous offre trois façons de boucler : `while`, `do while` et `for`.
- `while` s'utilise plutôt pour boucler tant qu'une condition n'est pas remplie alors que `for` est utilisé pour itérer sur un nombre connu d'éléments.
- Il faut faire attention à ne pas tomber dans une boucle infinie.
- Nous disposons de deux mots-clefs, `break` et `continue`, pour affiner notre contrôle sur l'exécution de la boucle.

Contenu masqué

Contenu masqué n°13 : Correction laverie

```
#include <iostream>

int main()
{
    std::cout <<
        "Bienvenue à la laverie de Clem. Combien de kilos de linge as-tu ? ";
    int kilos { 0 };
    std::cin >> kilos;

    if (kilos <= 5)
    {
        std::cout <<
            "Tu as peu de linge, mets-le dans une machine de 5kg."
            << std::endl;
    }
    else if (kilos <= 10)
    {
        std::cout <<
            "Tu as beaucoup de linge, mets-le dans une machine de 10kg."
            << std::endl;
    }
    else
    {
        // On a plus que 10 kilos, il va falloir diviser le linge
            en plusieurs machines.
        std::cout <<
            "Dites donc, il faut laver son linge plus souvent !" <<
            std::endl;

        int nb_machines_10_kilos { 0 };

        while (kilos > 5)
        {
            // Pour chaque machine de 10kg utilisée, on enlève 10kg
                au total de linge restant.
            kilos -= 10;
            ++nb_machines_10_kilos;
        }

        std::cout << "Tu as besoin de " << nb_machines_10_kilos <<
            " machine(s) de 10kg." << std::endl;
        if (kilos >= 0)
        {
```

```
34              // S'il reste entre 0 et 5 kilos, il ne faut pas
                    oublier la dernière machine de 5kg.
35              std::cout <<
                    "Le reste rentrera dans une machine de 5kg." <<
                    std::endl;
36          }
37      }
38
39      return 0;
40  }
```

Retourner au texte.

Contenu masqué n°14 : Correction PGCD

```
 1  #include <iostream>
 2
 3  int main()
 4  {
 5      std::cout << "Rentre un nombre a : ";
 6      int a { 0 };
 7      std::cin >> a;
 8
 9      std::cout << "Rentre un nombre b : ";
10      int b { 0 };
11      std::cin >> b;
12
13      int r { a % b };
14      while (r != 0)
15      {
16          a = b;
17          b = r;
18          // r vaut le reste de la division entière de a par b.
19          r = a % b;
20      }
21
22      std::cout << "PGCD vaut = " << b << std::endl;
23      return 0;
24  }
```

Retourner au texte.

Contenu masqué n°15 :

Correction somme de 1 à n

```
#include <iostream>

int main()
{
    std::cout << "Donne-moi un entier : ";
    int n { 0 };
    std::cin >> n;

    int total { 0 };
    int compteur { 1 };

    while (compteur <= n)
    {
        total += compteur;
        ++compteur;
    }

    std::cout << "La somme totale vaut " << total << "." <<
        std::endl;
    return 0;
}
```

Ceux qui aiment les maths ont remarqué qu'il s'agit de la somme des premiers entiers de 1 à n et qu'il existe une formule pour calculer directement sans boucler : $\frac{N \times (N+1)}{2}$. Bravo à vous.
Retourner au texte.

Contenu masqué n°16 :
Correction T.P partie II

```
#include <iostream>

int main()
{
    std::cout << "Quel jour es-tu né ? ";
    int jour { 0 };

    // Comme std::cin >> jour renvoie false s'il y a eu une erreur,
        afin de rentrer dans la boucle, on prend la négation de
        l'expression avec !
    while (!(std::cin >> jour))
    {
        std::cout << "Entrée invalide. Recommence." << std::endl;
        std::cin.clear();
```

```
13          std::cin.ignore(255, '\n');
14      }
15
16      std::cout << "Quel mois es-tu né ? ";
17      int mois { 0 };
18
19      // Même version avec un do while.
20      do
21      {
22          if (std::cin.fail())
23          {
24              std::cout << "Entrée invalide. Recommence." <<
                    std::endl;
25              std::cin.clear();
26              std::cin.ignore(255, '\n');
27          }
28
29      } while (!(std::cin >> mois));
30
31      std::cout << "Tu es né le " << jour << "/" << mois << "." <<
            std::endl;
32      return 0;
33  }
```

Les deux versions sont légèrement différentes dans ce qu'elles font. La première demande l'entrée, puis tant qu'elle est invalide, elle nettoie puis demande à nouveau. La deuxième vérifie s'il y a eu une quelconque erreur, la nettoie puis demande l'entrée et recommence si l'entrée est invalide.

Peu importe celle que vous avez choisi d'utiliser, l'essentiel est de nettoyer nos entrées et d'avoir des programmes robustes. Et c'est justement ce que nous faisons. Retourner au texte.

Contenu masqué n°17 : Indice

Nous allons devoir tester tous les nombres entre 2 et `x` pour savoir s'ils sont diviseurs de `x`. Mais si nous trouvons un nombre `y` différent de `x` et qui divise `x`, alors nous savons que le nombre n'est pas premier, inutile de continuer. Retourner au texte.

Contenu masqué n°18 : Correction

```
1  #include <iostream>
2
3  int main()
4  {
```

```
5       std::cout <<
            "Donne-moi un nombre, je te dirai s'il est premier : ";
6       int nombre { 0 };
7       std::cin >> nombre;
8
9       bool est_premier { true };
10      for (int i { 2 }; i < nombre; ++i)
11      {
12          if (nombre % i == 0)
13          {
14              // Si i divise nombre, alors nous savons que nombre
                    n'est pas premier.
15              est_premier = false;
16              break;
17          }
18      }
19
20      if (est_premier)
21      {
22          std::cout << nombre << " est un nombre premier !" <<
                std::endl;
23      }
24      else
25      {
26          std::cout << nombre << " n'est pas premier." << std::endl;
27      }
28
29      return 0;
30  }
```

Retourner au texte.

Contenu masqué n°19 : Indice

Pour rappel, un nombre est pair s'il est divisible par 2 et impair autrement. Retourner au texte.

Contenu masqué n°20 : Correction

```
1   #include <iostream>
2
3   int main()
4   {
```

```
5       std::cout <<
            "Donne-moi un nombre, je t'afficherai tous les nombres impairs jusqu'à
6       int maximum { 0 };
7       std::cin >> maximum;
8
9       for (int i { 0 }; i <= maximum; ++i)
10      {
11          if (i % 2 == 0)
12          {
13              // On saute ce nombre puisqu'il est pair.
14              continue;
15          }
16
17          std::cout << i << std::endl;
18      }
19
20      return 0;
21  }
```

Retourner au texte.

7. Au tableau !

Si je vous demande de me faire un programme calculant la moyenne d'une série de nombres, comment feriez-vous ? Avec une boucle, comme nous avons appris dans le chapitre précédent. Mais maintenant, j'ai besoin de les stocker. Comment faire, puisque vous ne savez pas à l'avance combien de notes je veux rentrer ?

Ce chapitre va résoudre cette problématique en vous présentant **les tableaux**.

7.1. Un tableau c'est quoi ?

Les tableaux, en informatique, font partie de la grande famille des **structures de données** . Ces structures de données sont des façons particulières **d'organiser et de stocker des données**. Il y a ainsi des structures pour stocker un nombre fixe de données du même type, d'autres pour un nombre variable, d'autre encore pour stocker des données de types différents. Ces objets, conçus spécialement pour stocker des données, sont appelés **conteneurs**.

Celles qui nous intéressent maintenant sont les tableaux. Ils sont conçus pour stocker des **données de même type** et ayant **des points communs** : notes d'un élève à l'école, titres de livres, âge d'utilisateurs. Dès que vous pensez « liste de... », dès que des éléments ont des points communs **et qu'il fait sens de les regrouper**, alors l'usage d'un tableau peut s'envisager.

Un tableau n'est cependant pas la solution universelle, mais ça, nous le verrons au fur et à mesure, pour ne pas vous encombrer inutilement de détails.

7.2. std : :vector — Mais quel dynamisme !

Commençons par les **tableaux dynamiques**, c'est-à-dire dont la taille ne sera connue qu'au moment de l'exécution du programme. C++ nous fournit justement un objet déjà codé pour ça et qui peut facilement s'agrandir ou se rétrécir : `std::vector`. Et le fichier d'en-tête correspondant se doit bien évidemment d'être inclus en faisant `#include <vector>`.

7.2.1. Déclarer un `std::vector`

Les concepteurs de `std::vector` l'ont créé de telle sorte qu'il soit utilisable avec n'importe quel type. On peut donc utiliser un `std::vector` de `int` pour stocker des âges et un `std::vector` de `double` pour stocker des moyennes. Voici la façon de faire.

```
std::vector</* type des éléments du tableau */> identifiant {};
```

Avec maintenant trois exemples concrets.

```
#include <string>
// N'oubliez pas cette ligne.
#include <vector>

int main()
{
    std::vector<int> const tableau_de_int {};
    std::vector<double> const tableau_de_double {};
    // Même avec des chaînes de caractères c'est possible.
    std::vector<std::string> const tableau_de_string {};

    return 0;
}
```

Tous nos `std::vector`, ou tableaux, sont vides avec ces déclarations. Mais on peut très bien leur affecter des valeurs, comme ci-dessous. Chacun de nos tableaux a un nombre d'éléments différents dès sa création.

```
#include <string>
#include <vector>

int main()
{
    // On stocke 5 entiers.
    std::vector<int> const tableau_de_int { 1, 2, 3, 4, 5 };
    // Et ici 2 flottants.
    std::vector<double> const tableau_de_double { 2.71828, 3.14159
        };
    // Maintenant 3 chaînes de caractères.
    std::vector<std::string> const tableau_de_string {
        "Une phrase.", "Et une autre !", "Allez, une pour la fin."
        };

    return 0;
}
```

Rien ne nous empêche non plus de copier un tableau dans un autre, tout comme on le fait pour des `int`.

```cpp
#include <vector>

int main()
{
    std::vector<int> const tableau_de_int { 1, 2, 3 };
    // Maintenant, copie possède les mêmes éléments que
        tableau_de_int.
    std::vector<int> const copie { tableau_de_int };

    return 0;
}
```

7.2.2. Manipuler les éléments d’un `std::vector`

8. Accès aux éléments

Pour l'instant, nos tableaux sont inutiles, nous ne faisons rien avec. Commençons donc à nous amuser un peu en récupérant un élément du tableau pour l'afficher. On utilise pour cela les crochets devant l'identifiant de notre tableau, avec entre crochets la position de l'élément à récupérer.

```
identificateur[/* un entier */];
```

```cpp
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> const nombres { 2, 4, 6, 8 };
    std::cout << "Deuxième élément: " << nombres[1] << '\n';

    return 0;
}
```

? Pourquoi récupère-t-on le deuxième élément en écrivant `1` ? Le deuxième élément, ça devrait être `2` non ?

Justement non. La subtilité des tableaux en C++, c'est qu'**ils commencent à l'indice 0 et non l'indice 1**. Pourquoi ça ? C'est un héritage du C que je ne vais pas détailler car il demande des connaissances que vous n'avez pas encore. Souvenez-vous simplement que **le premier élément d'un tableau est à l'indice 0** et que donc le $x$ème élément d'un tableau est à l'indice $x - 1$.

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> const tableau_de_int { 1, 2, 3 };

```

```
    std::cout << "Le premier élément est " << tableau_de_int[0] <<
        "." << std::endl;
    std::cout << "Le deuxième élément est " << tableau_de_int[1] <<
        "." << std::endl;
    std::cout << "Le troisième élément est " << tableau_de_int[2]
        << "."  << std::endl;

    return 0;
}
```

?

Et si j'accède à l'élément 4, que se passe-t-il ?

Potentiellement, tout et n'importe quoi. Parfois, le programme va continuer sans rien dire, d'autre fois vous allez avoir des erreurs et des plantages, comme je l'illustre ci-dessous. Parfois, le programme peut aussi péter complètement un câble et lancer un missile nucléaire sur un pays pris au hasard. C'est ce qu'on appelle un **comportement indéterminé**, en anglais « ***undefined behavior*** ». Impossible donc de faire du code de qualité dans ces conditions. Voilà pourquoi il ne faut pas utiliser d'indice inférieur à 0 ou plus grand que la taille de notre tableau moins un.

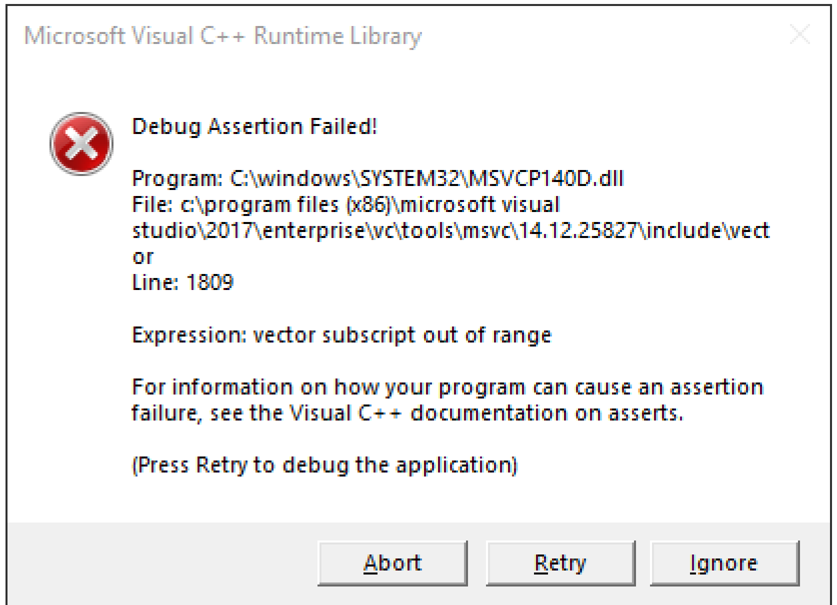


FIGURE 8.1. – Un exemple avec Visual Studio 2017 de ce qui arrive.

Comportement indéterminé

Un « UB » est le **signe d'une erreur de programmation**. Vous avez mal fait quelque chose, quelque part dans le code et le programme ne peut plus être considéré comme fiable désormais. Il faut donc à tout prix les éviter et faire attention en codant !

8.0.0.1. Premier et dernier élément

Il existe d'ailleurs deux fonctions dont le but est d'accéder spécifiquement au premier ou au dernier élément d'un tableau. Elles se nomment, respectivement, `front` et `back`. Leur utilisation est très simple.

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> const tableau_de_int { 1, 2, 3, 4, 5, 6, 7, 8,
        9 };

    std::cout << "Le premier élément est " <<
        tableau_de_int.front() << "." << std::endl;
    std::cout << "Le dernier élément est " << tableau_de_int.back()
        << "." << std::endl;

    return 0;
}
```

8.0.0.2. Obtenir la taille d'un tableau

Justement, comment récupère-t-on la taille de notre tableau ? En utilisant la fonction `std::size`, qui renvoie le nombre d'éléments de n'importe quel conteneur.

Le type de retour de cette fonction est un type nouveau, que nous n'avons pas rencontré jusque-là et qui s'appelle `std::size_t`. C'est un type entier conçu, entre autres, pour être capable de stocker la taille de n'importe quel tableau, aussi grand soit-il.

En vrai, un `int` est suffisant et `std::size_t` est présent car historiquement déjà là en C. De toute façon, avec `auto`, nous n'avons aucune raison de nous casser la tête là-dessus.

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> const tableau_de_int { 1, 2, 3 };

    auto const taille { std::size(tableau_de_int) };
    std::cout << "Mon tableau contient " << taille << " éléments."
        << std::endl;

    return 0;
```

```
12 }
```

8.0.0.3. Afficher les éléments d'un tableau

Comme nous l'avions dit dans le chapitre sur les boucles, depuis l'arrivée de C++11, il existe une variante de la boucle `for`, conçue exclusivement pour parcourir les éléments d'un conteneur. Ci-dessous, voici un pseudo-code illustrant cette possibilité.

```
1 for (/* type d'un élément du tableau ou auto */ identifiant : /*
      tableau à parcourir */)
2 {
3     // Manipuler identifiant, en l'affichant par exemple.
4 }
```

Voici ce que cela donne avec un exemple concret. Pas besoin de récupérer la taille, la boucle est intelligente. Cela nous facilite la vie et rend notre code plus concis et plus facile à lire.

```
1  #include <iostream>
2  #include <vector>
3
4  int main()
5  {
6      std::vector<int> const tableau_entiers { 1, 2, 3 };
7
8      std::cout << "Je vais afficher mon tableau en entier.\n";
9      for (auto const element : tableau_entiers)
10     {
11         std::cout << element << std::endl;
12     }
13
14     return 0;
15 }
```

i `auto` ou pas ?

Quant à utiliser `auto` ou préciser explicitement le type d'un élément (`int` par exemple), c'est une question personnelle. Certains ne vont jurer que par `auto`, d'autres ne le mettront que si le type est long à écrire. Vous êtes libres de choisir ce qu'il vous plaît.

8.0.0.4. Vérifier si un tableau est vide

Fonction bien utile, `std::empty` permet de vérifier si un tableau est vide, c'est-à-dire qu'il ne contient aucun élément et donc qu'il a une taille de zéro. Elle renvoie un booléen, donc cela signifie que cette fonction appliquée à un tableau vide renverra `true` et `false` pour un tableau avec trois éléments, par exemple.

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> const tableau_vide {};
    std::vector<int> const tableau_rempli { 1, 2, 3 };

    std::cout << std::boolalpha;
    std::cout << "Est-ce que tableau_vide est vide ? Réponse : " <<
        std::empty(tableau_vide) << std::endl;
    std::cout << "Est-ce que tableau_rempli est vide ? Réponse : "
        << std::empty(tableau_rempli) << std::endl;

    return 0;
}
```

8.0.0.5. Ajouter des éléments

Pour l'instant, nos `std::vector` ne sont pas trop dynamiques. Mais cela va changer avec la fonction `push_back`, qui va nous donner la possibilité d'ajouter un élément à la fin de notre tableau. Celui-ci doit cependant être du même type que les autres éléments du tableau. On ne pourra ainsi pas ajouter de `int` à un tableau de `std::string`.

```
#include <iostream>
#include <string>
#include <vector>

int main()
{
    std::vector<int> tableau_de_int { 12, 24 };
    // On ajoute un élément...
    tableau_de_int.push_back(36);
    // ...mais on peut en ajouter encore d'autres.
    tableau_de_int.push_back(48);
    tableau_de_int.push_back(100);

    // On affiche pour vérifier.
```

```
15      for (auto const valeur : tableau_de_int)
16      {
17          std::cout << valeur << std::endl;
18      }
19
20      std::vector<std::string> tableau_de_string { "Salut !",
            "Voici une phrase." };
21
            tableau_de_string.push_back("Mais je vais en ajouter une autre.");
22      // Ceci ne compilera pas.
23      //tableau_de_string.push_back(5);
24
25      for (auto const chaine : tableau_de_string)
26      {
27          std::cout << chaine << std::endl;
28      }
29
30      return 0;
31  }
```

Et pour vous prouver que la taille change bien, il suffit d'utiliser `std::size` comme vu plus haut.

```
 1  #include <iostream>
 2  #include <vector>
 3
 4  int main()
 5  {
 6      std::vector<int> tableau_de_int { 12, 24 };
 7
 8      std::cout << "Taille avant l'ajout : " << tableau_de_int.size()
            << std::endl;
 9
10      tableau_de_int.push_back(36);
11      tableau_de_int.push_back(48);
12      tableau_de_int.push_back(100);
13
14      std::cout << "Taille après l'ajout : " <<
            std::size(tableau_de_int) << std::endl;
15
16      return 0;
17  }
```

8.0.0.6. Supprimer des éléments

On ajoute, on ajoute, mais on a aussi envie de pouvoir retirer un voire tous les éléments. Cette possibilité nous est également offerte avec les fonctions `pop_back` et `clear`. Ces fonctions sont

simples, comme l'illustre l'exemple ci-dessous.

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> tableau_de_int { 12, 24 };

    std::cout << "Taille avant l'ajout : " <<
        std::size(tableau_de_int) << std::endl;

    tableau_de_int.push_back(36);
    tableau_de_int.push_back(48);
    tableau_de_int.push_back(100);

    std::cout << "Taille après l'ajout : " <<
        std::size(tableau_de_int) << std::endl;

    // Je retire le dernier élément.
    tableau_de_int.pop_back();
    std::cout << "Taille après avoir retiré le dernier élément : "
        << std::size(tableau_de_int) << std::endl;

    // Finalement j'en ai marre, je vide le tableau.
    tableau_de_int.clear();
    std::cout << "Taille après avoir vidé le tableau : " <<
        std::size(tableau_de_int) << std::endl;

    return 0;
}
```

Comportement indéterminé

Appeler `pop_back` sur un tableau vide est aussi un comportement indéterminé.

8.0.0.7. Modifier des éléments

Pour l'instant, nous ne faisons que lire les valeurs de notre tableau, mais il n'y a aucun problème pour les modifier, tant que le tableau n'est pas déclaré comme `const` bien évidemment. Et la bonne nouvelle, c'est que vous savez déjà comment faire : les crochets `[]`. En effet, s'ils sont suivis d'un `=` puis d'une valeur, alors l'élément du tableau auquel on accède se voit modifié.

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> tableau { 1, 2, 3 };

    // Finalement, je préfère que la première valeur soit un 15.
    tableau[0] = 15;

    for (auto const valeur : tableau)
    {
        std::cout << valeur << std::endl;
    }

    return 0;
}
```

De même que pour la lecture, n'essayez pas de modifier la valeur d'un élément en dehors du tableau. Je vous le rappelle, c'est un comportement indéterminé et qui est potentiellement dangereux !

8.0.0.8. Remplir le tableau avec une valeur unique

Dernière fonction que je vais vous montrer, celle-ci est bien utile pour remplir d'un coup un tableau d'une même valeur. La fonction `assign` a besoin de savoir combien de valeurs vous comptez insérer et quelle est-elle.

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> valeurs {};
    std::cout << "Taille actuelle de mon tableau : " <<
        std::size(valeurs) << " éléments.\n";

    // On ajoute dix fois la valeur 42.
    valeurs.assign(10, 42);
    std::cout << "Taille actuelle de mon tableau : " <<
        std::size(valeurs) << " éléments.\n";

    for (auto const valeur : valeurs)
    {
        std::cout << valeur << std::endl;
    }
```

```
17
18      std::cout << '\n';
19
20      // Finalement on réduit en asignant cinq fois la valeur 3.
21      valeurs.assign(5, 3);
22      std::cout << "Taille actuelle de mon tableau : " <<
            std::size(valeurs) << " éléments.\n";
23
24      for (auto const valeur : valeurs)
25      {
26          std::cout << valeur << std::endl;
27      }
28
29      std::cout << '\n';
30
31      // Mais on peut très bien augmenter.
32      valeurs.assign(12, 78);
33      std::cout << "Taille actuelle de mon tableau : " <<
            std::size(valeurs) << " éléments.\n";
34
35      for (auto const valeur : valeurs)
36      {
37          std::cout << valeur << std::endl;
38      }
39
40      return 0;
41  }
```

8.1. Exercices

8.1.1. Calcul de moyenne

Dans cet exercice, vous allez laisser l'utilisateur entrer autant de valeurs qu'il le souhaite puis en faire la moyenne, c'est-à-dire diviser la somme des notes par le nombre de notes. L'utilisateur pourra taper autant de valeurs positives qu'il le souhaite. Cependant, un nombre négatif signifiera qu'il ne souhaite plus entrer de note.

Correction calcul de moyenne

8.1.2. Minimum et maximum

Le but de ce programme est tout simple : il faut trouver le plus petit élément du tableau, ainsi que le plus grand. Pour se simplifier la vie, vous pouvez utiliser un tableau déjà rempli.

Correction maximum et minimum

8.1.3. Séparer les pairs des impairs

Cette fois, on veut créer deux nouveaux tableaux qui contiendront, pour le premier les nombres pairs uniquement, pour le second les nombres impairs.

Correction pairs et impairs

8.1.4. Compter les occurrences d'une valeur

Une opération très courante avec un tableau est de compter le nombre de fois que revient une certaine valeur. Faites donc un programme qui permet de compter le nombre de fois que revient une lettre dans un tableau de `char`.

Correction occurrence d'une valeur

8.2. std : :array — Le tableau statique

Abordons maintenant l'autre catégorie de tableaux qui existe en C++ : les **tableaux statiques**, c'est-à-dire dont la taille est **connue à la compilation** et **qui ne peut varier**. En contrepartie, ce tableau est plus performant et plus rapide qu'un `std::vector` puisqu'il n'y a pas d'opération d'ajout ou de retrait d'éléments.

Pour l'utiliser, il faut inclure le ficher d'en-tête `<array>`.

8.2.1. Déclarer un `std::array`

Comme pour `std::vector`, notre nouveau conteneur attend qu'on lui indique non seulement quel type de données il doit stocker, mais également **combien**. Je rappelle que la taille doit être **définie à la compilation**. Vous ne pouvez pas déclarer un `std::array` et lui passer une taille ultérieurement. Voyez vous-mêmes l'exemple ci-dessous.

```
1 #include <array>
2 #include <iostream>
3
4 int main()
5 {
```

```
        // Tableau contenant cinq entiers explicitement précisés.
        std::array<int, 5> const tableau_de_5_entiers { 1, 2, 3, 4,
            5 };
        for (auto const element : tableau_de_5_entiers)
        {
                std::cout << element << std::endl;
        }

        // Tableau contenant deux décimaux définis puis cinq zéros.
        std::array<double, 7> const tableau_de_7_decimaux {
            3.14159, 2.1878 };
        for (auto const element : tableau_de_7_decimaux)
        {
                std::cout << element << std::endl;
        }

        // Tableau qui a trop d'éléments et ne compilera pas.
        //std::array<int, 2> const tableau_qui_ne_compilera_pas {
            1, 2, 3, 4 };

        // Il est tout à fait possible de copier un tableau.
        auto copie { tableau_de_5_entiers };
        for (auto const element : copie)
        {
                std::cout << element << std::endl;
        }

    return 0;
}
```

Avez-vous remarqué que, contrairement à `std::vector`, un tableau statique défini avec `std::array` contient des éléments par défaut ? Ainsi, même si l'on ne précise pas tous les éléments, comme pour `tableau_de_7_decimaux`, nous avons la garantie que les autres seront initialisés à zéro.

8.2.2. Remplir un tableau

Il existe une fonction permettant de remplir le tableau, mais contrairement à `std::vector` qui peut être redimensionné et donc attend le nouveau nombre d'éléments, la fonction `fill` pour `std::array` n'attend que la valeur utilisée pour remplir le tableau.

```
#include <array>
#include <iostream>

int main()
{
```

```
    std::array<int, 5> valeurs { 1, 2, 3, 4, 5 };

    std::cout << "Avant" << std::endl;
    for (auto const valeur : valeurs)
    {
        std::cout << valeur << std::endl;
    }

    // On remplit de 42.
    valeurs.fill(42);

    std::cout << std::endl << "Après" << std::endl;
    for (auto const valeur : valeurs)
    {
        std::cout << valeur << std::endl;
    }

    return 0;
}
```

8.2.3. Accéder aux éléments

Comme précédemment, pour accéder à un élément du tableau, nous allons utiliser les crochets `[]`. Et comme pour `std::vector`, faites attention à **ne pas accéder à un élément en dehors des limites du tableau**.

```
#include <array>
#include <iostream>

int main()
{
    std::array<double, 7> tableau_de_7_decimaux { 3.14159, 2.1878
        };
    // C'est le nombre d'or.
    tableau_de_7_decimaux[2] = 1.61803;

    // Aïe aïe aïe, catastrophe !
    //tableau_de_7_decimaux[7] = 1.0;

    for (auto const element : tableau_de_7_decimaux)
    {
        std::cout << element << std::endl;
    }

    return 0;
}
```

Les fonctions `front` et `back` sont toujours de la partie.

```cpp
#include <array>
#include <iostream>

int main()
{
    std::array<int, 9> const tableau_de_int { 1, 2, 3, 4, 5, 6, 7,
        8, 9 };

    std::cout << "Le premier élément est " <<
        tableau_de_int.front() << "." << std::endl;
    std::cout << "Le dernier élément est " << tableau_de_int.back()
        << "." << std::endl;

    return 0;
}
```

8.2.4. Connaître la taille

Sans surprise, c'est la même fonction que pour `std::vector`, je parle de `std::size`. Le code qui suit ne devrait donc pas vous surprendre.

```cpp
#include <array>
#include <iostream>

int main()
{
    std::array<int, 5> const valeurs { 1, 2, 3, 4, 5 };
    std::cout << "Et la taille de mon tableau vaut bien " <<
        std::size(valeurs) << "." << std::endl;
    return 0;
}
```

8.2.5. Vérifier si le tableau est vide

De même que vu plus haut, rien de dépaysant, la fonction `std::empty` renvoie `false` si le tableau contient des éléments et `true` s'il est vide. En pratique, cette fonction ne vous renverra jamais `true` car avoir un tableau vide signifie ici que la taille donnée est `0`. Autant vous dire que c'est inutile d'avoir un tableau vide.

```cpp
#include <array>
#include <iostream>

int main()
{
    std::array<int, 5> const valeurs { 1, 2, 3, 4, 5 };
    std::cout << std::boolalpha;
    std::cout << "Est-ce que mon tableau est vide ? " <<
        std::empty(valeurs) << " Ah, je le savais !" << std::endl;
    return 0;
}
```

8.3. std : :string — Un type qui cache bien son jeu

Vous vous demandez peut-être toujours à quoi servent **vraiment** les tableaux. Vous voudriez un exemple **concret** d'un tableau qui rend de grands services. Ça tombe très bien puisque vous en utilisez un depuis le début de ce cours : `std::string`.

En effet, vous vous souvenez que C++ utilise le type `char` pour stocker un caractère. De là, stocker du texte se fait en stockant un ensemble de `char`. Le type `std::string` n'est donc rien d'autre qu'**un tableau de `char`**, conçu et optimisé spécialement pour le stockage de texte.

8.3.1. Connaître la taille d'une chaîne

En toute originalité, on peut récupérer la taille d'une `std::string` grâce à la fonction `std::size`.

```cpp
#include <cassert>
#include <iostream>
#include <string>

int main()
{
    std::string const phrase { "Voici une phrase normale." };
    std::cout << "Voici la longueur de cette phrase : " <<
        std::size(phrase) << " caractères." << std::endl;

    return 0;
}
```

8.3.2. Accéder à un caractère

Comme pour les deux conteneurs que nous avons découverts plus tôt dans ce chapitre, nous pouvons utiliser les crochets `[]` pour **accéder à un caractère** en particulier. Et, exactement comme pour les deux autres, il faut faire attention à bien utiliser un indice qui est inférieur à la taille de la chaîne.

```
#include <cassert>
#include <iostream>
#include <string>

int main()
{
    std::string const phrase { "Ceci me servira d'exemple." };
    auto const indice { 5 };

    std::cout << "Voici le caractère à la position " << indice <<
        " : " << phrase[indice] << std::endl;
    return 0;
}
```

On peut bien entendu modifier un caractère sans problème.

```
#include <iostream>
#include <string>

int main()
{
    std::string verbe { "Estamper" };
    verbe[3] = 'o';
    std::cout << "Verbe choisi : " << verbe << std::endl;

    return 0;
}
```

8.3.3. Premier et dernier caractère

Les deux fonctions `front` et `back` nous rendent le même service que précédemment, en nous permettant de récupérer le premier et le dernier caractère de la chaîne.

```
#include <iostream>
#include <string>

```

```
int main()
{
    std::string const phrase { "Bonjour tout le monde !" };
    std::cout << "Première lettre : " << phrase.front() <<
        std::endl;
    std::cout << "Dernière lettre : " << phrase.back() <<
        std::endl;

    return 0;
}
```

8.3.4. Vérifier qu'une chaîne est vide

Toujours sans surprise pour vous, nous allons utiliser la même fonction `std::empty` que pour `std::vector` et `std::array`.

```
#include <iostream>
#include <string>

int main()
{
    std::string const texte { "Du texte." };

    std::cout << std::boolalpha;
    std::cout << "Est ce que 'texte' est vide ? " <<
        std::empty(texte) << std::endl;

    std::string const rien { "" };
    std::cout << "Est ce que 'rien' est vide ? " <<
        std::empty(rien) << std::endl;

    return 0;
}
```

8.3.5. Ajouter ou supprimer un caractère à la fin

Nous bénéficions également des fonctions `push_back` et `pop_back` pour, respectivement, ajouter et supprimer un caractère à la fin de la chaîne. Faites toujours attention à ne pas retirer le dernier caractère d'une chaîne vide, c'est un comportement indéfini.

```
#include <iostream>
#include <string>
```

```
3
4 int main()
5 {
6     std::string phrase { "Une phrase !" };
7
8     phrase.pop_back();
9     phrase.pop_back();
10
11    phrase.push_back('.');
12
13    std::cout << phrase << std::endl;
14
15    return 0;
16 }
```

8.3.6. Supprimer tous les caractères

Nous avons une méthode portant le doux nom de `clear`, de l'anglais « débarrasser, nettoyer, vider », qui va nous laisser une chaîne vide.

```
1 #include <cassert>
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     std::string phrase { "Blablabla du texte." };
8     phrase.clear();
9     std::cout << "Rien ne va s'afficher : " << phrase << std::endl;
10
11    return 0;
12 }
```

8.3.7. Boucler sur une chaîne

On peut bien évidemment boucler sur tous les caractères qui composent une chaîne à l'aide de `for`.

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
```

```
 6      std::string const phrase { "Voici un ensemble de lettres." };
 7      for (char lettre : phrase)
 8      {
 9          std::cout << lettre << std::endl;
10      }
11
12      return 0;
13  }
```

Bien entendu, `std::string` est encore plus fourni que ça et permet de faire de nombreuses autres choses. Nous aurons l'occasion de les examiner plus tard dans ce tutoriel. Retenez simplement que **`std::string` peut être manipulé comme un tableau**.

8.3.8. En résumé

- Les tableaux nous permettent de stocker un ensemble de valeurs liées entre elles (notes, âges, moyennes, caractères, etc).
- Il existe plusieurs types de tableaux en C++ : dynamique quand la taille est connue à l'exécution et statique quand elle est connue à la compilation.
- `std::vector` est un tableau dynamique, dont le nombre d'éléments peut augmenter ou diminuer à volonté.
- `std::array` est un tableau statique, dont la taille est fixée définitivement à la compilation, et dont on ne peut pas ajouter ni supprimer d'éléments.
- `std::string` est un tableau dynamique spécialement conçu pour la gestion du texte.

Contenu masqué

Contenu masqué n°21 : Correction calcul de moyenne

```
 1  #include <iostream>
 2  #include <vector>
 3
 4  int main()
 5  {
 6      std::vector<double> notes {};
 7
 8      while (true)
 9      {
10          std::cout <<
                "Entre une note (inférieur à 0 pour arrêter) : ";
11          double note { 0.0 };
```

```
12
13                  // N'oublions pas de protéger notre entrée.
14          while (!(std::cin >> note))
15                  {
16                          std::cout << "Entrée invalide. Recommence."
                                << std::endl;
17                          std::cin.clear();
18                          std::cin.ignore(255, '\n');
19                  }
20
21          if (note < 0)
22          {
23              // On s'arrête si la note est inférieur à 0.
24              break;
25          }
26
27          // On ajoute la note à la liste des notes.
28          notes.push_back(note);
29      }
30
31      if (!std::empty(notes))
32      {
33          double total{};
34          for (auto const note : notes)
35          {
36              total += note;
37          }
38
39          std::cout << "Ta moyenne est de " << total / notes.size()
                << "." << std::endl;
40      }
41      else
42      {
43          std::cout <<
                "Tu n'as pas rentré de notes, je ne peux pas calculer ta moyenne."
                << std::endl;
44      }
45
46      return 0;
47  }
```

Aviez-vous pensé à prendre en compte si l'utilisateur ne rentre aucune note car sa seule entrée est négative ? Retourner au texte.

Contenu masqué n°22 :

Correction maximum et minimum

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> const entiers { 5, -2, 74, 455, -12 };
    int minimum { entiers[0] };
    int maximum { entiers[0] };

    for (auto const entier : entiers)
    {
        if (entier < minimum)
        {
            minimum = entier;
        }

        if (entier > maximum)
        {
            maximum = entier;
        }
    }

    std::cout << "Le minimum est " << minimum << "." << std::endl;
    std::cout << "Le maximum est " << maximum << "." << std::endl;

    return 0;
}
```

Retourner au texte.

Contenu masqué n°23 : Correction pairs et impairs

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> const entiers{ 5, -2, 74, 455, -12, 10, 11 };
    std::vector<int> pairs {};
    std::vector<int> impairs {};

    for (auto const entier : entiers)
```

```
11      {
12          if (entier % 2 == 0)
13          {
14              pairs.push_back(entier);
15          }
16          else
17          {
18              impairs.push_back(entier);
19          }
20      }
21
22      std::cout << "Les pairs sont : " << std::endl;
23      for (int const entier_pair : pairs)
24      {
25          std::cout << entier_pair << std::endl;
26      }
27
28      std::cout << std::endl << "Les impairs sont : " << std::endl;
29      for (int const entier_impair : impairs)
30      {
31          std::cout << entier_impair << std::endl;
32      }
33
34      return 0;
35  }
```

Retourner au texte.

Contenu masqué n°24 : Correction occurrence d'une valeur

```
1   #include <iostream>
2   #include <vector>
3
4   int main()
5   {
6       std::vector<char> const lettres { 'a', 'b', 'b', 'c', 'b', 'd',
          'd', 'a' };
7       char const lettre_a_compter { 'b' };
8       int total { 0 };
9
10      for (auto const lettre : lettres)
11      {
12          if (lettre == lettre_a_compter)
13          {
14              ++total;
```

```
15          }
16      }
17
18      std::cout << "La lettre '" << lettre_a_compter <<
            "' est présente " << total << " fois." << std::endl;
19
20      return 0;
21  }
```

Retourner au texte.

9. Déployons la toute puissance des conteneurs

Depuis que nous avons commencé ce cours, nous avons étudié deux conteneurs très utiles, `std::vector`, qui est un tableau dynamique, et `std::array`, un tableau statique. Nous avons appris qu'on peut accéder à un élément grâce aux crochets `[]`. Cette solution n'est pourtant **pas très générique**, puisque avec d'autres conteneurs C++, comme `std::list`, les crochets ne sont pas disponibles.

Ce chapitre va donc introduire une solution générique à ce problème qui porte le nom **d'itérateurs**, largement employés dans la bibliothèque standard, notamment avec les **algorithmes**.

9.1. Le socle de base : les itérateurs

Qu'est-ce qu'un itérateur ? Littéralement, il s'agit de quelque chose dont on se sert **pour itérer**. Dans notre cas, c'est sur des conteneurs que nous voulons itérer. Et qui dit concept de boucle, dit accès aux éléments un par un. Nous avons vu plus tôt les crochets `[]` avec `std::array` et `std::vector`. Il s'agit ici du même concept, mais encore plus puissant, car totalement **indépendant du conteneur utilisé** !

Un itérateur peut accomplir ceci puisque il sait à quel endroit de la séquence regarder. On dit que l'itérateur est **un curseur sur l'élément**. Et en se déplaçant d'emplacement en emplacement, il peut ainsi parcourir l'ensemble d'une collection de données.

9.1.1. Déclaration d'un itérateur

Pour déclarer un itérateur, rien de plus simple, vous connaissez déjà la moitié de la syntaxe. Schématiquement, le code ressemblera à ça. Notez que le type de l'identificateur doit correspondre à celui du conteneur associé : on ne peut pas utiliser un itérateur de `std::vector<double>` pour parcourir un `std::vector<char>`.

```
1 // Dans le cas d'un conteneur modifiable.
2 std::/* Le conteneur utilisé. */::iterator identificateur { /*
      L'élément sur lequel pointer. */ };
3 // Dans le cas d'un conteneur const.
4 std::/* Le conteneur utilisé. */::const_iterator identificateur {
      /* L'élément sur lequel pointer. */ };
```

Pour raccourcir le code et se simplifier la vie, on peut bien entendu faire appel à l'inférence de type, à l'aide d'`auto`. C'est ce que je vais faire dans le reste de ce cours.

```
auto identificateur { /* L'élément sur lequel pointer. */ };
```

9.1.2. Début et fin d'un conteneur

Notez, dans la déclaration précédente, le commentaire `/* L'élément sur lequel pointer. */`. Deux fonctions, dans un premier temps, vont nous intéresser : `std::begin`, un curseur vers le **premier élément** du conteneur, et `std::end`, un curseur vers un **élément virtuel inexistant**, qui se trouve **après le dernier élément**.

Pourquoi ne pas pointer vers le dernier élément de la collection plutôt ?

Pour une raison simple : comment faites-vous donc pour **déterminer que vous êtes à la fin de la collection** ? En effet, si `std::end` pointe vers l'adresse en mémoire d'un élément valide de la collection, qu'est-ce qui vous prouve que vous êtes à la fin ? C'est pour cela que `std::end` renvoie un itérateur sur un élément inexistant, qui indique la fin de la collection.

```
#include <vector>

int main()
{
    std::vector<int> const tableau { -1, 28, 346 };
    // Déclaration explicite d'un itérateur de std::vector
        d'entiers.
    std::vector<int>::const_iterator debut_tableau {
        std::begin(tableau) };
    // Déclaration plus courte en utilisant auto.
    auto fin_tableau { std::end(tableau) };

    return 0;
}
```

9.1.3. Accéder à l'élément pointé

Nous savons comment pointer vers le premier élément de la collection, ainsi que signaler sa fin, mais ça ne nous avance pas vraiment. Il nous manque l'accès aux éléments. Pour cela, il faut **déréférencer** l'itérateur, grâce à l'étoile `*` devant celui-ci. L'exemple suivant affiche la valeur sur laquelle pointe l'itérateur, c'est-à-dire le premier élément du tableau, ici `-1`.

```cpp
#include <iostream>
#include <string>

int main()
{
    std::string tableau { "Tu texte." };
    auto debut_tableau { std::begin(tableau) };

    // On affiche le premier élément du tableau.
    std::cout << *debut_tableau << std::endl;
    // Et on peut même le modifier, comme ce qu'on fait avec les
        références, ou les crochets.
    *debut_tableau = 'D';
    // On vérifie.
    std::cout << tableau << std::endl;

    return 0;
}
```

Attention avec `std::end` !

Ne tentez pas de déréférencer `std::end`! Vous accéderiez à un espace mémoire inconnu, ce qui entraînera un comportement indéterminé, donc tout et n'importe quoi potentiellement. Je répète, ne tentez pas d'accéder à la valeur pointée par `std::end`, ni de la modifier.

9.1.4. Se déplacer dans une collection

10. Déplacement vers l'élément suivant

Maintenant que nous sommes capables d'afficher le premier élément, il serait bien de faire de même avec les autres. Nous pouvons, à partir d'un itérateur sur un élément, récupérer l'adresse de l'élément suivant en appliquant l'opérateur d'incrémentation `++`.

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> tableau { -1, 28, 346 };
    auto debut_tableau { std::begin(tableau) };

    std::cout << *debut_tableau << std::endl;

    // On incrémente l'itérateur pour pointer sur l'élément
        suivant.
    ++debut_tableau;

    // On affiche le deuxième élément.
    std::cout << *debut_tableau << std::endl;

    return 0;
}
```

10.0.0.1. Utilisation concrète

Nous avons maintenant tout ce qu'il nous faut pour afficher l'ensemble d'une collection. L'exemple suivant vous montre comment parcourir un conteneur à l'aide d'une boucle `for` classique et d'itérateurs. Vous noterez l'utilisation d'un identifiant `it`, qui est l'équivalent de `i` quand nous avions vu les boucles.

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> tableau { -1, 28, 346, 84 };
```

```
    
    for (auto it { std::begin(tableau) }; it != std::end(tableau);
        ++it)
    {
        std::cout << *it << std::endl;
    }

    return 0;
}
```

10.0.0.2. Déplacement vers l'élément précédent

La plupart des conteneurs fournissent aussi l'opération inverse, c'est-à-dire récupérer l'adresse de l'élément précédent, grâce à l'opérateur de décrémentation `--`. Cet opérateur va, par exemple, nous permettre de récupérer le dernier élément d'un conteneur et l'afficher.

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> tableau { -1, 28, 346, 84 };
    auto iterateur_dernier_element { std::end(tableau) };

    // On récupère l'adresse de l'élément précédent.
    --iterateur_dernier_element;
    // Et on affiche pour vérifier.
    std::cout << *iterateur_dernier_element << std::endl;

    return 0;
}
```

10.0.0.3. Se déplacer de plusieurs éléments

Admettons que vous vouliez vous déplacer de plusieurs éléments, disons pour récupérer le cinquième. N'y a-t-il pas un autre moyen que d'utiliser cinq fois l'opérateur `++` ? La réponse dépend du conteneur que vous utilisez. En effet, en fonction de celui-ci, les itérateurs derrières ne proposent pas exactement les mêmes fonctionnalités.

Les conteneurs comme `std::vector`, `std::array` ou `std::string`, que nous avons découvert plut tôt dans ce tutoriel, sont ce qu'on appelle des **conteneurs à accès direct** (*random access* en anglais), c'est-à-dire qu'on peut **accéder directement à n'importe quel élément**. Cela est vrai en utilisant les crochets `[]` (c'est même à leur présence qu'on reconnait qu'un conteneur est à accès direct), et c'est également vrai pour les itérateurs. Ainsi, on peut directement écrire ceci quand on utilise un conteneur à accès direct.

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> tableau { -1, 28, 346, 84, -94, 31, 784 };
    // Notez le +4 pour accéder au cinquième élément, car comme
        avec les crochets, on commence à 0.
    auto iterateur_cinquieme_element { std::begin(tableau) + 4 };

    std::cout << *iterateur_cinquieme_element << std::endl;

    return 0;
}
```

Il existe d'autres conteneurs, comme `std::list`, qui sont dit **bidirectionnels**, parce qu'ils autorisent le déplacement vers un élément précédent ou suivant, mais pas plusieurs d'un coup. Si vous remplacez, dans le code précédent, `std::vector` par `std::list`, vous verrez qu'il ne compilera pas.

Plusieurs types d'itérateurs

Il existe, dans la bibliothèque standard, **cinq types** différents d'itérateurs. Nous en avons déjà vu deux ici. Nous aurons l'occasion de voir les trois autres au fur et à mesure que nous progresserons dans le cours.

10.0.1. `const` et les itérateurs

Le concept d'itérateur réclame des explications supplémentaires concernant l'utilisation de `const`. En effet, avec les itérateurs, **on distingue la constance de l'itérateur lui-même de la constance de l'objet pointé**. Pour parler plus concrètement, appliquer `const` à un itérateur rend celui-ci constant, c'est-à-dire qu'on ne peut pas, une fois assigné, le faire pointer sur autre chose. Par contre, la valeur pointée est tout à fait modifiable.

```cpp
#include <string>
#include <iostream>

int main()
{
    std::string mot { "Donjour" };

    // Itérateur constant sur un élément modifiable.
    auto const it { std::begin(mot) };

```

```
    // Ceci est impossible.
    // it = std::end(mot);

    // Ceci est parfaitement autorisé.
    *it = 'B';

    std::cout << mot << std::endl;
    return 0;
}
```

Pour garantir la constance des éléments pointés, on va utiliser deux fonctions très proches de celles vues plus haut. Il s'agit de `std::cbegin` et de `std::cend`, pour *const begin* et *const end*. L'exemple suivant vous montre la différence.

```
#include <string>

int main()
{
    std::string mot { "Donjour" };

    // Itérateur constant sur un élément modifiable.
    auto const iterateur_constant { std::begin(mot) };

    // Ceci est impsosible.
    // it = std::end(mot);

    // Ceci est parfaitement autorisé.
    *iterateur_constant = 'B';

    auto iterateur_non_constant { std::cbegin(mot) };

    // Ceci est impossible.
    // *iterateur_non_constant = 'D';

    // Ceci est parfaitement autorisé.
    iterateur_non_constant = std::cend(mot) - 1;

    return 0;
}
```

On peut, bien entendu, combiner les deux formes pour obtenir un itérateur constant sur un élément constant.

10.0.2. Itérer depuis la fin

Il existe encore une troisième paire de fonctions, qui vont nous autoriser à parcourir une collection de la fin vers le début. Elles se nomment `std::rbegin` (*reverse begin*) et `std::rend` (*reverse*

end). Elles s'utilisent exactement comme les précédentes, leur utilisation est transparente.

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> const tableau { -1, 28, 346, 84 };

    // Affiche depuis la fin, en commençant par 84.
    for (auto it { std::rbegin(tableau) }; it !=
        std::rend(tableau); ++it)
    {
        std::cout << *it << std::endl;
    }

    return 0;
}
```

Et comme pour les autres, les itérateurs peuvent être déclarés comme constant avec `std::crbegin` et `std::crend`.

> ? Pourquoi avoir des itérateurs spéciaux ? Pourquoi ne pas partir de `std::end` et itérer jusqu'à `std::begin` ?

Pour vous répondre, je vais utiliser une illustration qu'a créé @gbdivers dans son cours C++, comme la licence m'y autorise.

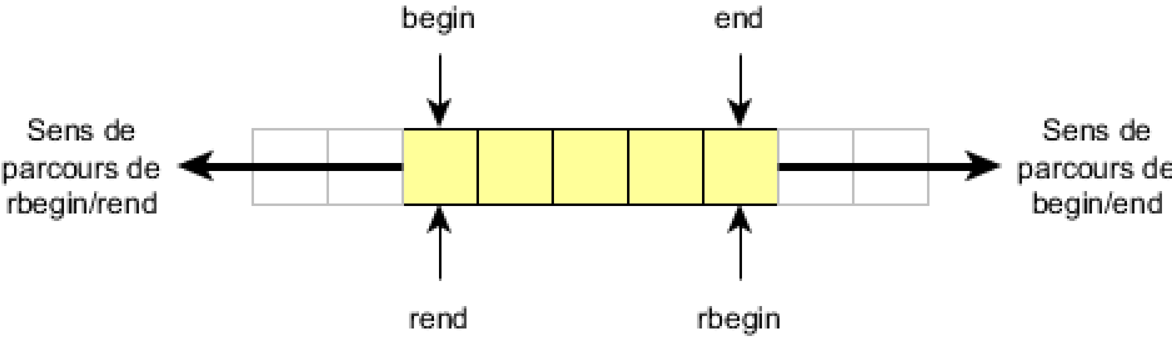


Figure 10.1. – Illustration des itérateurs classiques et *reverses*.

Si l'on part de `std::end`, on va se déplacer vers la droite et jamais l'on ne retombera sur `std::begin` ; le programme plantera bien avant. Voilà pourquoi il existe des itérateurs spécialement conçus pour parcourir une collection de la fin vers le début.

Alors, pourquoi ne pas partir de la fin et aller vers le début en décrémentant ?

Vous pourriez… mais ça serait tellement plus pénible. Il faudrait gérer les cas spéciaux que sont la valeur de début et la valeur de fin. Ou bien réécrire tous les algorithmes pour prendre en compte la décrémentation. Ou bien… utiliser `std::rbegin` et `std::rend` de manière totalement transparente.

10.0.3. Utilisation conjointe avec les conteneurs

Terminons cette section en présentant une possibilité d'initialisation des conteneurs qui nous était inconnue jusque-là. Grâce aux itérateurs, nous pouvons facilement **créer des sous-ensembles à partir d'une collection d'éléments**, comme un tableau qui ne contiendrait que les éléments 2 à 5 d'un autre tableau. Cela se fait très simplement : il suffit de donner en argument, entre accolades `{}`, l'itérateur de début et l'itérateur de fin.

```
1  #include <iostream>
2  #include <vector>
3
4  int main()
5  {
6      std::vector<int> collection { 1, -8, 4, 654, -7785, 0, 42 };
7      // On passe l'itérateur de début et celui de fin.
8      std::vector<int> sous_ensemble { std::begin(collection) + 1,
           std::begin(collection) + 5 };
9
10     // On affiche pour vérifier.
11     for (auto element : sous_ensemble)
12     {
13         std::cout << element << std::endl;
14     }
15
16     return 0;
17 }
```

Certains conteneurs, comme `std::vector`, `std::list` ou encore `std::string`, proposent également une fonction nommée `erase`, qui se base sur le principe des itérateurs pour supprimer plusieurs éléments d'une collection. On peut ainsi supprimer tous les éléments à partir du cinquième jusqu'à la fin, ou bien les trois premiers.

```
1  #include <algorithm>
2  #include <iostream>
3  #include <string>
4  #include <vector>
```

```
5
6  int main()
7  {
8      std::vector<int> nombres { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
9      // Certains éléments.
10     nombres.erase(std::begin(nombres) + 2, std::begin(nombres) +
           5);
11
12     for (auto nombre : nombres)
13     {
14         std::cout << nombre << std::endl;
15     }
16
17     std::cout << std::endl;
18
19     std::string phrase { "Voici une phrase beurk !" };
20     // Jusqu'à la fin.
21     phrase.erase(std::begin(phrase) + 16, std::end(phrase));
22     std::cout << phrase << std::endl;
23
24     return 0;
25 }
```

10.1. Des algorithmes à gogo !

Maintenant, les itérateurs vont devenir vraiment intéressants et concrets, parce qu'ils sont pleinement employés par **les algorithmes** de la bibliothèque standard. Ces derniers, assez nombreux, sont des **fonctions déjà programmées et fortement optimisées pour manipuler des collections**, permettant ainsi de les trier, de récupérer tous les éléments satisfaisant une certaine condition, etc. Nous n'en verrons que quelques-uns dans ce chapitre, mais une fois le principe compris, vous n'aurez aucune difficulté à utiliser les autres.

Fichier à inclure

La plupart des algorithmes se trouvent dans le fichier d'en-tête `<algorithm>`, pensez donc à l'inclure.

10.1.1. `std::count` — Compter les occurrences d'une valeur

Commençons avec un algorithme simple, mais on ne peut plus utile : compter les occurrences d'une certaine valeur dans une collection. Cela se fait avec l'algorithme `std::count`. Celui-ci attend trois paramètres.

— Le premier est un itérateur pointant sur le début de la séquence.
— Le deuxième est un itérateur pointant sur la fin de la séquence.

— Le dernier est l'élément à compter.

La force de l'algorithme, c'est qu'il ne nous impose pas de chercher dans toute la collection. En effet, comme on manipule ici des itérateurs, rien ne nous empêche de compter les occurrences d'une valeur seulement entre le troisième et le septième élément de la collection, ou entre le cinquième et le dernier.

```
#include <algorithm>
#include <iostream>
#include <string>

int main()
{
    std::string const phrase {
        "Exemple illustrant le tutoriel C++ de Zeste de Savoir." };

    // Toute la collection.
    auto const total_phrase { std::count(std::begin(phrase),
        std::end(phrase), 'e') };
    std::cout << "Dans la phrase entière, il y a " << total_phrase
        << " 'e' minuscule." << std::endl;

    // Un sous-ensemble seulement de la collection.
    auto const total_premier_mot { std::count(std::begin(phrase),
        std::begin(phrase) + 7, 'e') };
    std::cout << "Dans le premier mot, il y a " <<
        total_premier_mot << " 'e' minuscule." << std::endl;

    return 0;
}
```

Cet exemple utilise `std::string`, mais j'aurais tout aussi bien pu employer `std::vector` ou `std::array`. Par contre, `std::list` ne convient pas parce que … oui ! j'ai fait `std::begin(phrase) + 7`, ça suit dans le fond. Heureusement, une solution existe, que je vais vous montrer de suite.

10.1.2. `std::find` — Trouver un certain élément

Vous remarquerez que le code précédent n'est pas de très bonne qualité, puisque j'ai écrit `+7` directement, en dur. En plus du problème soulevé précédemment (ne pas pouvoir utiliser `std::list`), cela n'est pas très flexible : si le premier mot devient plus long ou plus court, je fausse les résultats. Alors qu'avec un itérateur qui pointerait sur le premier espace, on serait assuré d'avoir plus de souplesse. C'est justement ce que propose `std::find`, qui prend trois paramètres.

— Le premier est un itérateur pointant sur le début de la séquence.
— Le deuxième est un itérateur pointant sur la fin de la séquence.
— Le dernier est l'élément à rechercher.

Cette fonction retourne un **itérateur sur le premier élément correspondant à la recherche**, ou bien le deuxième argument si rien n'est trouvé.

```
#include <algorithm>
#include <iostream>
#include <string>

int main()
{
    std::string const phrase {
        "Exemple illustrant le tutoriel C++ de Zeste de Savoir." };

    // On obtient un itérateur pointant sur le premier espace
        trouvé.
    // Si l'on n'avait rien trouvé, on aurait obtenu
        std::end(phrase) comme valeur de retour.
    auto const iterateur_premier_mot {
        std::find(std::begin(phrase), std::end(phrase), ' ') };

    // On peut donc s'en servir ici. Même si l'on modifie la
        longueur du premier mot, on est assuré que cette solution
        marche.
    auto const total_premier_mot { std::count(std::begin(phrase),
        iterateur_premier_mot, 'e') };
    std::cout << "Dans le premier mot, il y a " <<
        total_premier_mot << " 'e' minuscule." << std::endl;

    return 0;
}
```

Je précise de nouveau pour être bien clair : `std::find` retourne un itérateur sur **le premier** élément trouvé, pas le deuxième ou le trousillième. Mais avec une simple boucle, rien ne nous empêche de récupérer la deuxième occurrence, ou la troisième.

Cela me donne d'ailleurs une excellente idée d'exercice. En utilisant les deux fonctions que nous venons de voir, nous allons faire un programme qui compte le nombre de `'e'` dans chaque mot d'une phrase. Essayez de coder par vous-mêmes ce petit exercice. Une solution est proposée ci-dessous.

Un indice

Petite astuce avant de commencer : on peut initialiser une chaîne de caractères avec un itérateur de début et de fin, comme n'importe quel autre conteneur.

Contenu masqué n°25

10.1.3. `std::sort` — Trier une collection

L'algorithme `std::sort`, fonctionnant, comme les autres, avec une paire d'itérateurs, permet de trier dans l'ordre croissant une collection d'éléments, que ce soit des nombres, des caractères, etc. Bien entendu, cet algorithme ne marchera que sur des conteneurs modifiables, c'est-à-dire non `const`. Voici un exemple tout simple qui ne devrait guère vous surprendre.

```
#include <algorithm>
#include <iostream>
#include <vector>

int main()
{
    std::vector<double> constantes_mathematiques { 2.71828, 3.1415,
        1.0836, 1.4142, 1.6180 };
    std::sort(std::begin(constantes_mathematiques),
        std::end(constantes_mathematiques));

    for (auto constante : constantes_mathematiques)
    {
        std::cout << constante << std::endl;
    }

    return 0;
}
```

Si vous avez essayé de changer `std::vector` en `std::list`, vous vous êtes rendu compte que le code ne compile plus. En effet, `std::sort` attend en paramètre des itérateurs à accès direct, comme ceux de `std::array`, `std::vector` et `std::string`.

À l'opposé, `std::list` n'utilise que des itérateurs bidirectionnels. Il est donc impossible de trier une `std::list` en utilisant cet algorithme.

Le cas de `std::list`

Le type `std::list` possède une fonction `sort` qui lui est propre et qui fait la même chose.

```cpp
#include <iostream>
#include <list>

int main()
{
    std::list<int> liste { 4, -8, 45, 2 };
    liste.sort();

    for (int i : liste)
    {
        std::cout << i << std::endl;
    }
    return 0;
}
```

10.1.4. `std::reverse` — Inverser l'ordre des éléments d'une collection

Parlons de `std:::reverse`, qui inverse la position des éléments compris entre l'itérateur de début et l'itérateur de fin. Un exemple va vous montrer clairement ce qu'il fait.

```cpp
#include <algorithm>
#include <iostream>
#include <list>

int main()
{
    std::list<int> nombres { 2, 3, 1, 7, -4 };
    std::cout << "Avant :\n";
    for (auto nombre : nombres)
    {
        std::cout << nombre << std::endl;
    }

    std::reverse(std::begin(nombres), std::end(nombres));

    std::cout << "\nAprès :\n";
    for (auto nombre : nombres)
    {
        std::cout << nombre << std::endl;
    }

    return 0;
}
```

10.1.5. `std::remove` — Suppression d'éléments

Admettons maintenant que vous voulez supprimer toutes les lettres `'b'` d'une chaîne. Avec `std::remove`, il n'y a aucun problème à ça. L'algorithme va supprimer toutes les occurrences de la lettre dans la chaîne. Et comme l'algorithme ne peut pas modifier la taille du conteneur, il nous renvoit un itérateur qui nous signale la fin de la collection. Il ne reste plus qu'à supprimer le tout grâce à `erase`.

```
1  #include <algorithm>
2  #include <iostream>
3  #include <string>
4
5  int main()
6  {
7      std::string paroles { "I'm blue, da ba dee da ba daa !" };
8
9      auto iterateur_fin { std::remove(std::begin(paroles),
           std::end(paroles), 'b') };
10     paroles.erase(iterateur_fin, std::end(paroles));
11
12     std::cout << paroles << std::endl;
13     return 0;
14 }
```

10.1.6. `std::search` — Rechercher un sous-ensemble dans un ensemble

Cette fois, avec `std::search`, ça n'est pas un unique élément que nous allons chercher, mais bien un sous-ensemble dans un autre ensemble, comme, par exemple, un mot dans une phrase. Cette fois, pas moins de quatre arguments sont attendus.

— Le premier est l'itérateur de début de la collection à fouiller.
— Le deuxième est l'itérateur de fin.
— Le troisième est l'itérateur de début de la sous-collection à rechercher.
— Le quatrième est l'itérateur de fin.

Tout comme `std::find`, si rien n'est trouvé, le deuxième argument, l'itérateur de fin de la collection à parcourir, est renvoyé. Voici donc un exemple pour illustrer le fonctionnement de l'algorithme.

```
1  #include <algorithm>
2  #include <iostream>
3  #include <string>
4
5  int main()
6  {
```

```cpp
    std::string const phrase {
        "Un exemple de phrase avec plusieurs mots." };
    auto const debut { std::begin(phrase) };
    auto const fin { std::end(phrase) };

    // Rappel : std::boolalpha permet d'afficher 'true' ou 'false'
        et non '1' ou '0'.
    std::cout << std::boolalpha;

    std::string mot { "mot" };
    std::cout << (std::search(debut, fin, std::begin(mot),
        std::end(mot)) != fin) << std::endl;

    mot = "exemmple";
    std::cout << (std::search(debut, fin, std::begin(mot),
        std::end(mot)) != fin) << std::endl;

    return 0;
}
```

10.1.7. `std::equal` — Vérifier l'égalité de deux ensembles

Comment testeriez-vous que deux ensembles sont équivalents, autrement dit qu'ils possèdent les mêmes éléments ? La réponse vient du titre : avec l'algorithme `std::equal`. Celui-ci prend plusieurs paramètres.

- L'itérateur de début du premier ensemble.
- L'itérateur de fin du premier ensemble.
- L'itérateur de début du deuxième ensemble.
- L'itérateur de fin du deuxième ensemble.

Si les deux ensembles n'ont pas la même taille, alors l'égalité est forcément impossible. Sinon, l'algorithme va comparer les deux ensembles élément par élément.

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> const premier { 1, 2, 3, 4 };
    std::vector<int> const second { 1, 2, 3, 4, 5 };
    std::vector<int> const troisieme { 1, 2, 3, 4 };

    std::cout << std::boolalpha;
    std::cout << std::equal(std::begin(premier), std::end(premier),
        std::begin(second), std::end(second)) << std::endl;
```

```
13      std::cout << std::equal(std::begin(premier), std::end(premier),
            std::begin(troisieme), std::end(troisieme)) << std::endl;
14
15  }
```

10.1.8. Et tant d'autres

Il y a tellement d'algorithmes différents que je ne vais pas tous vous les présenter. Vous en avez déjà vu plusieurs, et tous fonctionnent sur le même principe, celui **des itérateurs**. Utiliser les autres ne devrait donc pas vous poser de problème.

10.2. Personnalisation à la demande

Jusqu'ici, nos algorithmes sont figés. Par exemple, `std::sort` ne trie que dans l'ordre croissant. Mais les concepteurs de la bibliothèque standard ont pensé à ça et ont écrit leurs algorithmes de telle manière qu'on peut personnaliser leur comportement. Mine de rien, cette possibilité les rend encore plus intéressants! Voyons ça ensemble.

10.2.1. Le prédicat, à la base de la personnalisation des algorithmes

Un prédicat est **une expression qui prend, ou non, des arguments et renvoie un booléen**. Le résultat sera donc `true` ou `false`, si la condition est vérifiée ou non. Littéralement, un prédicat répond à des questions ressemblant à celles ci-dessous.

- Est-ce que l'élément est impair ?
- Est-ce que les deux éléments `a` et `b` sont égaux ?
- Est-ce que les deux éléments sont triés par ordre croissant ?

La très grande majorité des algorithmes utilisent les prédicats, même ceux vu plus haut, sans que vous le sachiez. Par exemple, `std::sort` utilise, par défaut, le prédicat `<`, qui compare si un élément `a` est strictement inférieur à un autre élément `b`, ce qui explique pourquoi l'algorithme trie une collection dans l'ordre croissant. Mais il est tout à fait possible de lui indiquer de faire autre chose.

La bibliothèque standard contient déjà quelques prédicats, regroupés dans le fichier d'en-tête `<functional>`. Nous apprendrons, plus tard dans ce tutoriel, à écrire nos propres prédicats.

10.2.2. Trier une liste dans l'ordre décroissant

Faisons connaissance avec `std::greater`. Il s'agit d'un **foncteur**, c'est-à-dire **un objet qui agit comme une fonction**. Les détails internes ne nous intéressent pas pour l'instant, nous y reviendrons plus tard, quand nous aurons les connaissances requises.

Ce foncteur compare deux éléments et retourne `true` si le premier est plus grand que le deuxième, sinon `false`. Voici quelques exemples. Notez qu'on doit préciser, entre chevrons `<>`, le type des objets comparés.

```
#include <functional>
#include <iostream>

int main()
{
    std::cout << std::boolalpha;
    // On peut l'utiliser directement.
    std::cout << std::greater<int>{}(2, 3) << std::endl;
    std::cout << std::greater<int>{}(6, 3) << std::endl;
    std::cout << std::greater<double>{}(0.08, 0.02) << std::endl;

    // Ou bien à travers une variable, qu'on appelle comme une
        fonction.
    std::greater<int> foncteur {};
    std::cout << foncteur(-78, 9) << std::endl;
    std::cout << foncteur(78, 8) << std::endl;

    return 0;
}
```

Afin de personnaliser le comportement de `std::sort`, notre fonction de tri vue plus haut, il suffit de lui passer ce foncteur. Observez par vous-mêmes.

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> tableau { -8, 5, 45, -945 };
    std::sort(std::begin(tableau), std::end(tableau),
        std::greater<int> {});

    for (auto x : tableau)
    {
        std::cout << x << std::endl;
    }

    return 0;
}
```

10.2.3. Somme et produit

Dans le fichier d'en-tête `<numeric>`, il existe une fonction répondant au doux nom de `std::accumulate`, qui permet de faire la somme d'un ensemble de valeurs, à partir d'une valeur de départ. Ça, c'est l'opération par défaut, mais on peut très bien lui donner un autre prédicat, comme `std::multiplies`, afin de calculer le produit des éléments d'un ensemble.

```
#include <functional>
#include <iostream>
#include <numeric>
#include <vector>

int main()
{
    std::vector<int> const nombres { 1, 2, 4, 5, 10 };

    // On choisit 0 comme élément de départ pour calculer la somme,
        car 0 est l'élément neutre de l'addition.
    auto somme { std::accumulate(std::cbegin(nombres),
        std::cend(nombres), 0) };
    std::cout << somme << std::endl;

    // On choisit 1 comme élément de départ pour calculer le
        produit, car 1 est l'élément neutre de la multiplication.
    auto produit { std::accumulate(std::cbegin(nombres),
        std::cend(nombres), 1, std::multiplies<int> {}) };
    std::cout << produit << std::endl;

    return 0;
}
```

10.2.4. Prédicats pour caractères

Il existe, dans le fichier d'en-tête `<cctype>`, des prédicats vérifiant des conditions **uniquement sur un caractère**. Ils permettent de vérifier si un caractère est une lettre, un chiffre, un signe de ponctuation, etc. Voyez par vous-mêmes.

```
#include <cctype>
#include <iostream>

int main()
{
    char const lettre { 'A' };

```

```
8       std::cout << "Est-ce que " << lettre << " est une minuscule ? "
            << islower(lettre) << std::endl;
9       std::cout << "Est-ce que " << lettre << " est une majuscule ? "
            << isupper(lettre) << std::endl;
10      std::cout << "Est-ce que " << lettre << " est un chiffre ? " <<
            isdigit(lettre) << std::endl;
11
12      char const chiffre { '7' };
13
14      std::cout << "Est-ce que " << chiffre << " est un chiffre ? "
            << isdigit(chiffre) << std::endl;
15      std::cout << "Est-ce que " << chiffre <<
            " est un signe de ponctuation ? " << ispunct(chiffre) <<
            std::endl;
16
17      return 0;
18  }
```

Particularité héritée

Ces fonctions sont **héritées du C**, c'est pour cela qu'elles ne sont pas précédées de l'habituel `std::`. C'est également la raison pour laquelle elles renvoient des entiers et non des booléens, comme on aurait pu s'y attendre.

On peut faire des choses intéressantes avec ça. Par exemple, en utilisant l'algorithme `std::all_of`, qui vérifie que tous les éléments d'un ensemble respectent une propriété, on peut vérifier si tous les caractères d'une chaîne sont des chiffres, pour valider qu'il s'agit d'un numéro de téléphone.

```
1   #include <algorithm>
2   #include <cctype>
3   #include <iostream>
4   #include <string>
5
6   int main()
7   {
8       std::string const numero { "0185017204" };
9       if (std::all_of(std::cbegin(numero), std::cend(numero),
            isdigit))
10      {
11          std::cout << "C'est un numéro de téléphone correct." <<
                std::endl;
12      }
13      else
14      {
15          std::cout << "Entrée invalide." << std::endl;
16      }
17
```

```
18        return 0;
19    }
```

Nos propres prédicats

Plus loin dans le cours, nous apprendrons à écrire nos propres prédicats, ce qui nous donnera encore plus de pouvoir sur nos algorithmes. Mais patience.

10.3. Exercices

10.3.1. Palindrome

Vous devez tester si une chaîne de caractères est un palindrome, c'est-à-dire un mot pouvant être lu indifféremment de gauche à droite ou de droite à gauche. Un exemple : « kayak. »

Correction palindrome

10.3.2. `string_trim` — Suppression des espaces

Allez, un exercice plus dur, pour vous forger le caractère. Le but est de reproduire une fonction qui existe dans de nombreux autres langages, comme C# ou Python, et qui permet de supprimer les espaces, au début et à la fin d'une chaîne de caractères. Cela signifie non seulement les espaces `' '` mais aussi les tabulations `'\t'`, les retours à la ligne `'\n'`, etc. Pour vous aider, vous aller avoir besoin de deux choses.

— Le prédicat `isspace`, pour savoir si un caractère est un type d'espace ou non.
— L'algorithme `std::find_if_not`, qui permet de trouver, dans un ensemble délimité par deux itérateurs, le premier caractère qui ne respecte pas une condition.

Pour vous faciliter la vie, découper l'exercice en deux. Faites d'abord la fonction permettant de supprimer tous les espaces à gauche, puis pour ceux à droite. Bon courage.

Correction `string_trim`

10.3.3. Couper une chaîne

Une autre opération courante, et qui est fournie nativement dans d'autres langages comme Python ou C#, consiste à découper une chaîne de caractères selon un caractère donné. Ainsi, si je coupe la chaîne `"Salut, ceci est une phrase."` en fonction des espaces, j'obtiens en résultat `["Salut,", "ceci", "est", "une", "phrase.]`.

Le but est donc que vous codiez un algorithme qui permet de faire ça. Vous allez notamment avoir besoin de la fonction `std::distance`, qui retourne la distance entre deux itérateurs, sous forme d'un nombre entier.

Correction découpage de chaînes

10.3.4. En résumé

- Les itérateurs sont des abstractions représentant un pointeur sur un élément d'un conteneur. Ils rendent ainsi l'usage du conteneur plus transparent.
- Deux fonctions, `std::begin` et `std::end`, sont extrêmement utilisées avec les itérateurs, car retournant un pointeur sur, respectivement, le début et la fin d'une collection.
- Les itérateurs demandent de la rigueur pour ne pas manipuler des éléments qui sont hors de la collection, ce qui entrainerait des comportements indéterminés.
- La bibliothèque standard contient des algorithmes, déjà programmés et très optimisés, qu'on peut appliquer à des collections.
- Nous pouvons les personnaliser grâce à des prédicats fournis dans la bibliothèque standard.

Contenu masqué

Contenu masqué n°25

```
#include <algorithm>
#include <iostream>
#include <string>

int main()
{
    std::string const phrase {
        "Exemple illustrant le tutoriel C++ de Zeste de Savoir, mais un peu plu
        };

    // Pour garder en mémoire le début de chaque mot.
    auto iterateur_precedent { std::begin(phrase) };
```

```cpp
    auto iterateur_espace { std::find(std::begin(phrase), std::end(phrase), ' ') };

    // Tant qu'on est pas à la fin de la phrase.
    while (iterateur_espace != std::end(phrase))
    {
        std::string const mot { iterateur_precedent, iterateur_espace };
        auto const total_e { std::count(std::begin(mot), std::end(mot), 'e') };

        std::cout << "Dans le mot '" << mot << "', il y a " << total_e << " fois la lettre 'e'." << std::endl;

        // On incrémente pour ne pas garder l'espace au début, car iterateur_espace pointe déjà sur un espace.
        ++iterateur_espace;
        // On met à jour notre itérateur de sauvegarde.
        iterateur_precedent = iterateur_espace;
        // On cherche la première occurrence dans le nouveau sous-ensemble.
        iterateur_espace = std::find(iterateur_espace, std::end(phrase), ' ');
    }

    // Une dernière fois pour analyser les caractères restants.
    std::string const dernier_mot { iterateur_precedent, std::end(phrase) };
    std::cout << "Dans le dernier mot '" << dernier_mot << "', il y a " << std::count(iterateur_precedent, std::end(phrase), 'e') << " fois la lettre 'e'." << std::endl;

    return 0;
}
```

Retourner au texte.

Contenu masqué n°26 : Correction palindrome

```cpp
#include <algorithm>
#include <iostream>
#include <string>

int main()
```

```
6  {
7      std::string const s1 { "azerty" };
8      std::string const s2 { "radar" };
9  
10     std::cout << std::boolalpha;
11     std::cout << std::equal(std::begin(s1), std::end(s1),
           std::rbegin(s1), std::rend(s1)) << std::endl;
12     std::cout << std::equal(std::begin(s2), std::end(s2),
           std::rbegin(s2), std::rend(s2)) << std::endl;
13 
14     return 0;
15 }
```

L'astuce consiste à regarder si les deux ensembles de lettres sont les mêmes, en partant de la gauche vers la droite pour le premier (itérateur classique) et l'opposé pour le deuxième (*reverse iterator*). Retourner au texte.

Contenu masqué n°27 : Correction `string_trim`

```
1  #include <algorithm>
2  #include <cctype>
3  #include <iostream>
4  #include <string>
5  
6  int main()
7  {
8      std::string test { "\n\tHello !\n\t" };
9      std::cout << "Avant modification : " << test << std::endl;
10 
11     // On utilise l'itérateur fourni par std::find_if_not, qui
           pointe sur le premier élément qui n'est pas un espace.
12     auto premier_non_espace { std::find_if_not(std::begin(test),
           std::end(test), isspace) };
13     test.erase(std::begin(test), premier_non_espace);
14     // On affiche pour tester.
15     std::cout << "Suppression au début : " << test << std::endl;
16 
17     // En inversant, tous les espaces de fin se retrouvent au
           début.
18     std::reverse(std::begin(test), std::end(test));
19     // On reprend le même algorithme.
20     premier_non_espace = std::find_if_not(std::begin(test),
           std::end(test), isspace);
21     test.erase(std::begin(test), premier_non_espace);
22     // On revient à l'état initial.
```

```
23      std::reverse(std::begin(test), std::end(test));
24      // On affiche pour tester.
25      std::cout << "Suppression à la fin : " << test << std::endl;
26
27      return 0;
28  }
```

Il est vrai que nous répétons deux fois les instructions pour supprimer les espaces à gauche. Laissez ça en l'état, nous verrons très bientôt une solution pour corriger ce fâcheux problème.
Retourner au texte.

Contenu masqué n°28 : Correction découpage de chaînes

```
1   #include <algorithm>
2   #include <iostream>
3   #include <string>
4   #include <vector>
5
6   int main()
7   {
8       std::string texte { "Voici une phrase que je vais couper." };
9       char const delimiteur { ' ' };
10      std::vector<std::string> parties {};
11
12      auto debut = std::begin(texte);
13      auto fin = std::end(texte);
14      auto iterateur = std::find(debut, fin, delimiteur);
15
16      while (iterateur != fin)
17      {
18          // Grâce à std::distance, nous obtenons la taille du mot.
19          std::string mot { debut, debut + std::distance(debut,
                iterateur) };
20
21          parties.push_back(mot);
22          // +1 pour sauter le délimiteur.
23          debut = iterateur + 1;
24          // Et on recommence.
25          iterateur = std::find(debut, fin, delimiteur);
26      }
27
28      // Ne pas oublier le dernier mot.
29      std::string mot { debut, debut + std::distance(debut,
            iterateur) };
30      parties.push_back(mot);
```

```
31
32      // On affiche pour vérifier.
33      for (std::string mot : parties)
34      {
35          std::cout << mot << std::endl;
36      }
37
38      return 0;
39  }
```

Retourner au texte.

11. Des flux dans tous les sens

Depuis le début de ce tutoriel, toutes les données que nous manipulons ont un point commun, qui est **leur volatilité**. Elles sont en effet stockées dans la mémoire vive de l'ordinateur, et celle-ci se vide dès qu'on l'éteint. Impossible, dans ces conditions, de sauvegarder des informations, comme une sauvegarde de jeu vidéo ou un résumé sur l'utilisateur.

Le but de ce chapitre est donc d'apprendre à **lire et écrire dans des fichiers** et, de manière plus générale, **dans des flux**.

Fichier à inclure

Pour manipuler des fichiers, il faut inclure le fichier d'en-tête `<fstream>`.

11.1. Avant-propos

Votre ordinateur contient énormément de fichiers, de tous les genres. Vos musiques préférées sont des fichiers, tout comme vos photos de vacances, votre CV Word ou LibreOffice, les fichiers de configuration de vos jeux vidéos, etc. Mêmes les programmes que vous utilisez sont des fichiers !

- Chaque fichier a **un nom**.
- Chaque fichier a **une extension**.

11.1.1. Prendrez-vous une extension ?

Il existe tout un tas d'extension de fichiers, pour chaque type qui existe. En voici quelques exemples.

- Fichiers audios : `.mp3`, `.oga`, etc.
- Fichiers vidéos : `.mp4`, `.avi`, etc.
- Images : `.jpg`, `.png`, `.gif`, etc.
- Exécutables : `.exe`, `.msi`, `.out`, etc.
- Fichiers textes : `.txt`, `.ini`, `.log`, etc.

Celles-ci sont surtout utiles pour nous, humains, afin de savoir quel type de fichier nous allons manipuler. Mais elles n'influencent en rien le contenu du fichier.

Certains fichiers, comme les fichiers textes, sont très simples à ouvrir et donne accès directement à l'information. D'autres, comme les `.png` ou les `.docx`, doivent être lus en respectant une certaine façon de faire, une norme, afin que l'information fasse sens. Dans le cadre de ce cours, **nous ne verrons que les documents textes simples**.

11.1.2. Vois sur ton chemin…

Afin de faciliter leur localisation et leur gestion, les fichiers sont classés et organisés sur leur support suivant un **système de fichiers**. C'est lui qui permet à l'utilisateur de répartir ses fichiers dans une arborescence de dossiers et de localiser ces derniers à partir d'un **chemin d'accès**.

Un chemin d'accès permet d'indiquer où est situé un fichier dans ledit système de fichier. Il contient forcément le nom du fichier concerné, mais également, si nécessaire, un ou plusieurs noms de dossiers, qu'il est nécessaire de traverser pour accéder au fichier depuis **la racine**.

La racine d'un système de fichier est le point de départ de l'arborescence des fichiers et dossiers. Sous GNU/Linux, il s'agit du répertoire `/` tandis que sous Windows, chaque lecteur est une racine (comme `C:` par exemple). Si un chemin d'accès commence par la racine, alors celui-ci est dit **absolu** car il permet d'atteindre le fichier depuis n'importe qu'elle position dans l'arborescence. Si le chemin d'accès ne commence pas par la racine, il est dit **relatif** et ne permet de parvenir au fichier que depuis un point précis dans la hiérarchie de fichiers.

Prenons un exemple. Nous avons un fichier `code_nucléaire.txt` qui se trouve dans un dossier `top_secret`, lui-même dans `documents`, qui se trouve à la racine. Le chemin absolu ressemblera à ça.

- Windows : `C:\documents\top_secret\code_nucléaire.txt`
- GNU/Linux : `/documents/top_secret/code_nucléaire.txt`

Par contre, si nous sommes déjà dans le dossier `documents`, alors nous pouvons nous contenter d'utiliser un chemin relatif. Ce chemin ne marchera cependant plus si nous sortons du dossier `documents`.

- Windows : `top_secret\code_nucléaire.txt`
- GNU/Linux : `top_secret/code_nucléaire.txt`

Le point `.` représente le dossier courrant, c'est-à-dire `documents` dans notre exemple.

11.1.3. Un mot sur Windows

Depuis les années 1990, Windows supporte les chemins utilisant les slashs `/`. Si vous ouvrez l'explorateur de fichier et que vous écrivez `C:/Windows/System32`, ça marchera sans problème. Vous avez donc deux possibilités.

- Utilisez les slashs `/` si vous voulez que votre programme puisse fonctionner sur tous les systèmes.
- Utilisez les antislashs `\` si votre programme n'est destiné qu'à Windows.

Si vous décidez d'utiliser ces derniers, n'oubliez pas de les doubler en écrivant `\\`, car `\` seul est un caractère d'échappement. Ou bien utilisez les chaînes brutes.

11.1.4. Pas d'interprétation, on est des brutes

Utiliser les séquences d'échappement peut alourdir le code, en plus d'être casse-pieds à écrire. C++ offre une solution alternative qui se nomme **chaînes brutes** (de l'anglais *raw strings*). Par défaut, une chaîne brute commence par `R"(` et termine par `)"`. Tout ce qui se trouve entre ces deux séquences n'est pas interprété, mais tout simplement **ignoré**.

```cpp
#include <iostream>

int main()
{
    std::cout <<
        R"(Elle a dit "Va dans le dossier C:\Program Files" et regarde.)"
        << std::endl;
    return 0;
}
```

Et si jamais notre chaîne contient le délimiteur de fin, on peut personnaliser les délimiteurs pour qu'ils soient différents. Pour ça, on ajoute les caractères qu'on souhaite entre les guillemets et les parenthèses.

```cpp
#include <iostream>

int main()
{
    // Oups, ça ne marche pas...
    //std::cout << R"(Il a écrit "f(x)" au tableau.)" << std::endl;

    // Heureusement, je peux choisir moi-même mes délimiteurs.
    // Ici, tout ce qui est entre "&( et )&" n'est pas interprété.
    std::cout << R"&(Il a écrit "f(x)" au tableau.)&" << std::endl;
    return 0;
}
```

11.2. std : :ofstream — Écrire dans un fichier

Commençons par apprendre à écrire dans un fichier. Cela nous est utilise si nous voulons sauvegarder un score, les paramètres de configuration d'une application ou n'importe quel résultat de notre programme. Et pour ce faire, nous allons utiliser le type `std::ofstream`, abréviation de « *output file stream* », ce qui signifie « flux fichier de sortie ».

11.2.1. Ouvrir le fichier

Il attend en argument le nom du fichier à ouvrir. Par défaut, si le fichier n'existe pas, il est créé.

```
#include <fstream>

int main()
{
    // Je veux ouvrir un fichier nommé 'sortie.txt', qui se trouve
        dans le dossier du projet.
    // S'il existe, il sera ouvert. Sinon, il sera d'abord créé
        puis ouvert.
    std::ofstream fichier { "sortie.txt" };
    return 0;
}
```

11.2.2. Écriture

Pour écrire dedans, c'est très simple, il suffit de réutiliser les chevrons, comme avec `std::cout`. Voyez par vous-mêmes.

```
#include <fstream>

int main()
{
    std::ofstream fichier { "sortie.txt" };
    // On écrit un 3, un espace et un 4.
    fichier << 3 << " " << 4;

    return 0;
}
```

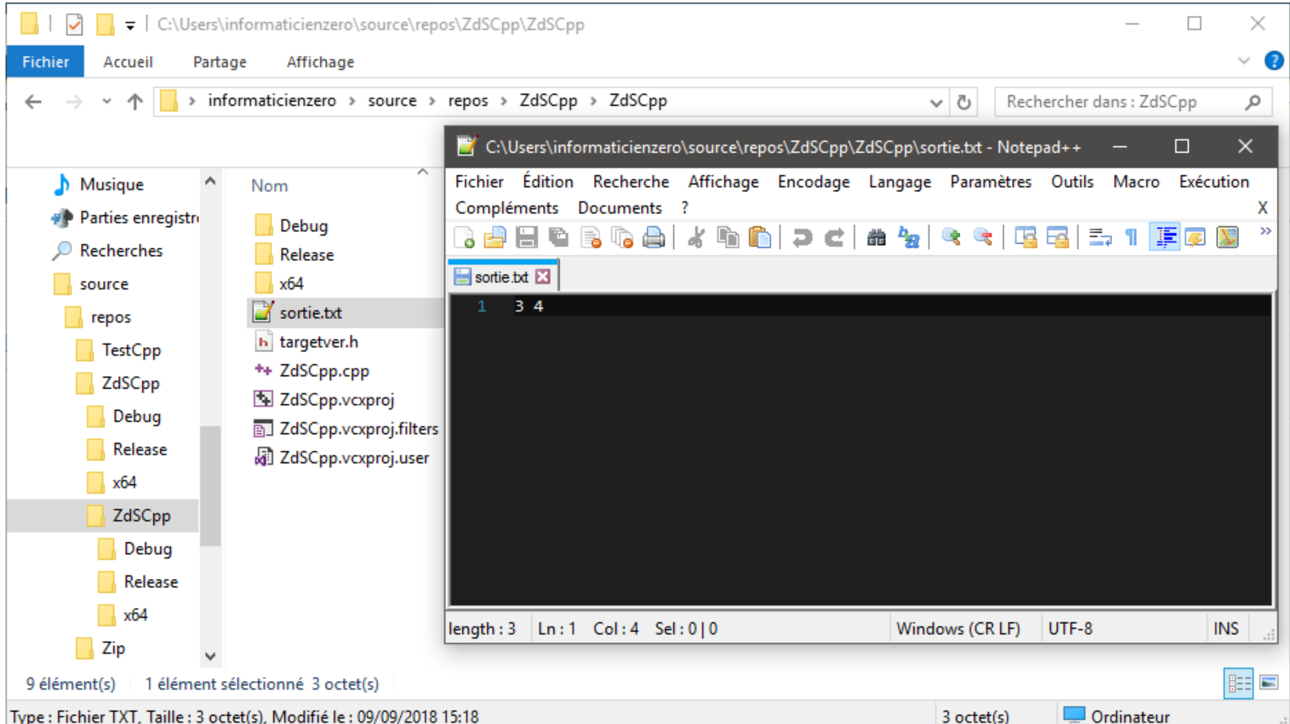


Figure 11.1. – On voit que les deux valeurs, ainsi que l'espace, ont bien été écrites.

Et, exactement comme pour `std::cout`, nous pouvons écrire des littéraux, mais aussi des variables et des expressions.

```cpp
#include <fstream>
#include <string>

int main()
{
    std::ofstream fichier { "sortie.txt" };
    fichier << 3 << " " << 4;

    int x { 0 };
    // On va à la ligne puis on écrit une équation.
    fichier << '\n' << x << " + 2 = " << x + 2;

    // Pas de problème avec le texte non plus.
    std::string texte { "Voici une phrase." };
    fichier << '\n' << texte;

    return 0;
}
```

J'ai essayé d'écrire une phrase avec des accents et j'ai obtenu un texte avec des caractères bizarres. Que s'est-il passé ?

Normalement, les Linuxiens sont épargnés par ce problème et seuls ceux utilisant Windows sont concernés (pas de chance). En fait, c'est une histoire d'**encodage**.

Sans rentrer dans les détails, car un cours entier existe déjà sur le sujet, il existe différents encodages qui permettent de stocker plus ou moins de caractères. Le premier inventé (années 1960), l'ASCII, ne permet de stocker que 127 caractères, dont les 26 lettres de l'alphabet anglais, mais pas nos accents français, le ñ espagnol, le ß allemand, etc. D'autres sont apparus et permettent de stocker plus de caractères, dont UTF-8 .

Par défaut en C++, quand on manipule une chaîne de caractères, celle-ci est interprétée comme étant de l'ASCII. Comme nos accents français ne sont pas représentables, on obtient du grand n'importe quoi en sortie. La solution consiste à utiliser UTF-8, ce qui se fait en précisant `u8` devant nos chaînes de caractères.

```
#include <fstream>
#include <iostream>
#include <string>
#include <vector>

int main()
{
    std::vector<std::string> const phrases
    {
        u8"Voici un mot important.\n",
        u8"Russe : резиновые перчатки",
        u8"Polonais : gumowe rękawiczki",
        u8"Roumain : mănusi de cauciuc"
    };

    std::ofstream fichier { "sortie.txt" };
    for (auto const & phrase : phrases)
    {
        fichier << phrase << std::endl;
    }

    return 0;
}
```

Enfin, il faut enregistrer le code source en UTF-8. Pour se faire, suivez les instructions ci-dessous.

— Sous Visual Studio, cliquez sur Fichier -> Enregistrer main.cpp sous…, puis cliquez sur la petite flèche à côté du bouton Enregistrer et sélectionnez l'option Enregistrer avec encodage…. Ensuite, pour le champs « Encodage », choisissez la valeur « **Unicode (UTF-8 avec signature) - Page de codes 65001** » et validez.

Il ne vous reste plus qu'à lancer et voir le résultat.

FIGURE 11.2. – Hop, plus de soucis, ça marche maintenant.

11.2.3. Ouvrir sans effacer

Essayer de modifier le fichier vous-mêmes, avec votre éditeur de texte préféré, et ajouter au tout début une phrase quelconque. Lancez le programme, rouvrez le fichier et horreur ! votre texte a disparu.

En fait, par défaut, `std::ofstream` tronque le contenu du fichier. Autrement dit, s'il y avait des données écrites précédemment, elles sont perdues dès que le programme rouvrira le fichier. Mais il est possible de régler ce problème très simplement.

Lorsque nous initialisons notre objet `std::ofstream`, nous pouvons lui donner un deuxième argument qui est le mode d'ouverture. Dans notre cas, ce mode est `std::ios::app`. Ici, `ios` veut dire *Input Output Stream*, c'est-à dire *flux d'entrée et de sortie* ; `app` est le diminutif de *append*, qui veut dire ajouter en anglais.

Notre contenu n'est plus effacé et les données supplémentaires sont *ajoutées* à la suite.

```
#include <fstream>
#include <iostream>
#include <string>

int main()
{
```

```
 7     std::ofstream fichier { "sortie.txt", std::ios::app };
 8     fichier << 3 << " " << 4;
 9
10     int x { 0 };
11     fichier << '\n' << x << " + 2 = " << x + 2;
12
13     std::string texte { u8"Voici une phrase." };
14     fichier << '\n' << texte;
15
16     return 0;
17 }
```

11.3. std : :ifstream — Lire dans un fichier

Abordons maintenant la lecture, opération elle aussi ô combien importante. Cela nous permet de charger des paramètres, d'analyser des fichiers, etc. Pour ce faire, nous allons utiliser le type `std::ifstream`, abréviation de «*input file stream*», ce qui signifie «flux fichier d'entrée».

11.3.1. Ouvrir le fichier

L'ouverture se fait de la même façon que pour `std::ofstream`, c'est-à-dire en donnant le nom du fichier à ouvrir. Sauf que cette fois, il faut que **le fichier existe déjà**. C'est logique : vous ne pouvez pas lire un fichier qui n'existe pas.

Créez donc avec moi un fichier appelé `entrée.txt`.

```
1 #include <fstream>
2
3 int main()
4 {
5     std::ifstream fichier { "entrée.txt" };
6     return 0;
7 }
```

11.3.2. Lecture

Écrivez avec moi, dans le fichier `entrée.txt`, un entier, puis un retour à la ligne, puis un mot.

```
1 42
2 Bonjour
```

La lecture se fait très simplement. On utilise les chevrons comme pour `std::cin`. Voyez par vous-mêmes.

```
#include <fstream>
#include <iostream>
#include <string>

int main()
{
    std::ifstream fichier { "entrée.txt" };

    int entier { 0 };
    fichier >> entier;
    std::cout << "Mon entier vaut : " << entier << std::endl;

    std::string mot { "" };
    fichier >> mot;
    std::cout << "Mon mot vaut : " << mot << std::endl;

    return 0;
}
```

?

J'avais dans mon fichier une phrase entière, et seul le premier mot est apparu dans la console. Que s'est-il passé?

En fait, c'est un comportement tout à fait normal, qu'on retrouve également avec `std::cin`. Par défaut, les espaces et les retours à la ligne sont considérés comme étant des **délimiteurs**, des **séparateurs**. Ainsi, là où vous voyez une phrase complète, le programme va voir un ensemble de mots séparés par des espaces.

Les concepteurs de C++ ne nous laissent néanmoins pas sans solution. Il existe une fonction, `std::getline`, qui permet de lire une ligne entière. Elle attend le flux d'entrée à lire, ainsi qu'une `std::string` pour écrire le résultat.

```
#include <fstream>
#include <iostream>
#include <string>

int main()
{
    std::ifstream fichier { "entrée.txt" };

    int entier { 0 };
    fichier >> entier;
    std::cout << "Mon entier vaut : " << entier << std::endl;
```

```
12
13      std::string phrase { "" };
14      std::getline(fichier, phrase);
15      std::cout << "Ma phrase vaut : " << phrase << std::endl;
16
17      return 0;
18  }
```

?

Ça ne marche toujours pas ! Pourquoi je ne vois plus rien maintenant ?

Vicieux comme je suis, je vous ai encore fait tomber dans un piège.

Le problème vient de l'utilisation, juste avant `std::getline`, des chevrons. Ceux-ci lisent la valeur entière, pas de soucis, mais **le caractère de retour à la ligne est laissé**. Si l'on refait encore cette opération avec les chevrons, alors ce caractère est « mangé » et ne nous gène pas. Mais ce n'est pas le cas avec `std::getline`.

Pour régler ce problème, nous avons plusieurs solutions.

— Nous pouvons utiliser `fichier.ignore(255, '\n')` après l'utilisation des chevrons, comme nous l'avons appris dans les T.P de gestion des erreurs d'entrée. Ainsi, nous sommes sûrs de supprimer le retour à la ligne qui nous embête.
— Nous pouvons utiliser le **modificateur de flux** (c'est-à-dire qui modifie son comportement) `std::ws` (de l'anglais « *white spaces* », espaces blancs). Il supprime tous les espaces et retours à la ligne qu'il trouve et s'utilise comme suit.

```
1  // On peut l'utiliser directement, en même temps que la lecture.
2  int entier { 0 };
3  fichier >> entier >> std::ws;
4  std::cout << "Mon entier vaut : " << entier << std::endl;
5
6
7  // On peut en faire une instruction à part.
8  fichier >> std::ws;
9
10
11 // Ou même directement avec std::getline !
12 std::getline(fichier >> std::ws, phrase);
```

Maintenant, en complétant le code précédent, nous n'avons plus aucun problème.

```
1  #include <fstream>
2  #include <iostream>
3  #include <string>
4
```

```cpp
5  int main()
6  {
7      std::ifstream fichier { "entrée.txt" };
8
9      int entier { 0 };
10     fichier >> entier;
11     std::cout << "Mon entier vaut : " << entier << std::endl;
12
13     std::string phrase { "" };
14     // On est sûr de ne pas oublier en l'utilisant directement dans
           std::getline.
15     std::getline(fichier >> std::ws, phrase);
16     std::cout << "Ma phrase vaut : " << phrase << std::endl;
17
18     return 0;
19 }
```

11.3.3. Tout lire

Vous êtes un lecteur vorace et vous voulez lire tout le fichier ? Aucun problème. Tant les chevrons (mot par mot) que `std::getline` (ligne par ligne) peuvent être utilisés dans une boucle, pour lire jusqu'à la fin du fichier.

```cpp
1  #include <fstream>
2  #include <iostream>
3  #include <string>
4
5  int main()
6  {
7      std::ifstream fichier { "entrée.txt" };
8      std::string ligne { "" };
9
10     while (std::getline(fichier, ligne))
11     {
12         std::cout << "Ligne lue : " << ligne << std::endl;
13     }
14
15     return 0;
16 }
```

Remarquez la syntaxe de la boucle: `while (std::getline(fichier, ligne))`. Cela devrait vous rappeler notre utilisation de `std::cin` dans les boucles (et plus généralement dans les conditions). Ce parallèle est dû au fait que la fonction `std::getline` renvoie un flux ; plus précisément, elle renvoie le flux qu'on lui donne en paramètre (ici, le fichier ouvert), dont l'état a donc été modifié. C'est ce flux qui est alors converti en booléen pour nous dire si tout s'est bien

passé. En particulier, lorsqu'on a atteint la fin du fichier, le flux renvoyé par `std::getline` est évalué à `false`, ce qui permet de sortir de la boucle en fin de lecture de fichier.

11.4. Exercice

11.4.1. Statistiques sur des fichiers

Quand vous écrivez avec un logiciel de traitement de texte, celui-ci vous donne en direct certaines informations, telles que le nombre de caractères, de mots, de paragraphes, etc. Faisons donc un petit programme qui va ouvrir un fichier et obtenir les informations suivantes.

- Le nombre de lignes.
- Le nombre de caractères (sans les espaces).
- Le nombre de mots.

Correction statistiques sur des fichiers

11.5. Encore plus de flux !

Nous savons lire et écrire dans des fichiers, ce qui est déjà bien. Mais allons un peu plus loin, voulez-vous ? Nous allons découvrir que la manipulation de fichiers et celle des entrées / sorties ont beaucoup plus de points communs que ce que vous auriez pu penser. En effet, dans les deux cas, on parle de **flux**.

11.5.1. Un flux c'est quoi ?

Il existe de très nombreux composants électroniques. Que ce soient des processeurs, des cartes-mères, des claviers, des écrans, des téléphones, une multitude s'offre à nous. Et tous viennent avec leur propre façon de communiquer. Écrire un programme qui devrait savoir communiquer avec tous ces périphériques est impossible. C'est pour cela qu'on utilise **des abstractions**.

Par exemple, votre **système d'exploitation** (Windows, GNU/Linux, Android, etc) est une abstraction. Notre programme ne communique plus avec un nombre potentiellement très grand de matériels et composants, mais uniquement avec le système d'exploitation, qui se charge lui-même de **faire le lien avec le matériel** (la mémoire vive, le processeur, le disque dur, etc). Cela nous simplifie grandement la vie.

La **bibliothèque standard** est aussi une abstraction. Elle nous fournit un lot unique de fonctionnalités précises, qui marcheront **peu importe le système d'exploitation** utilisé. Ainsi, en utilisant la bibliothèque standard, notre programme pourra être compilé pour fonctionner tant avec Windows, que GNU/Linux, qu'Android, etc.

Les flux sont eux aussi une abstraction. Grâce à eux, on peut envoyer ou recevoir des « flux de données » (d'où le nom), d'une taille potentiellement infinie, à des composants. Dans la

bibliothèque standard, nous avons ainsi de quoi recevoir des informations du clavier (`std::cin`), d'en envoyer à l'écran (`std::cout`) mais aussi avec des fichiers (`std::ifstream` et `std::ofstream`).

L'avantage, c'est qu'ils offrent une façon **transparente** et **unifiée** d'envoyer et de recevoir des données. Vous l'avez vu vous-mêmes plus haut, il n'y a aucune différence entre écrire sur la sortie standard ou un fichier de notre choix. On pourrait même coder nos propres flux de façon à lire et écrire sur le réseau, qui s'utiliseraient de la même façon que les autres !

11.5.2. Un *buffer* dites-vous ?

Les flux sont, selon le terme anglophone largement employé, *bufferisé*. Le problème est qu'accéder à des composants externes, des périphériques, comme la carte graphique, un appareil externe ou autre, peut être **long**. Ainsi, dans le code suivant, on accèderait potentiellement trois fois à un composant externe.

```
1 flux << 1 << 'A' << 3.1415;
```

Pour éviter ça, tout ce qu'on envoie ou reçoit d'un flux n'est pas directement traité mais mis en mémoire dans un tampon (en anglais « *buffer* »). Pour que les données soient concrètement et physiquement envoyées à l'écran, ou écrites sur un fichier ou autre, il faut **vider le tampon** (en anglais « *flush* »). C'est notamment ce que fait `std::endl`, en plus d'afficher un retour à la ligne. C'est également ce qui se passe quand le fichier est fermé, quand notre objet arrive à la fin de sa portée et est détruit.

Tenez, faisons un test. Compilez le code suivant et, avant de taper une phrase quelconque en guise d'entrée, ouvrez le fichier concerné et regardez son contenu. Vous le trouverez vide, car les données sont toujours dans le *buffer*.

```
 1 #include <fstream>
 2 #include <iostream>
 3 #include <string>
 4 
 5 int main()
 6 {
 7     std::ofstream fichier { "sortie.txt" };
 8     // Notez que je ne vide pas le tampon.
 9     fichier << "Hey, salut toi !\n";
10     fichier << 42 << " " << 2.718;
11 
12     // Une entrée pour vous laisser le temps d'ouvrir le fichier
          sortie.txt.
13     std::string phrase { "" };
14     std::cout << "Tape une phrase quelconque : ";
15     std::cin >> phrase;
16 
```

```
17      // Je vide explicitement le tampon. Maintenant les données sont
            écrites.
18      fichier << std::flush;
19      return 0;
20  }
```

Pour éviter ce phénomène parfois coûteux, certains codes remplacent `std::endl` par `'\n'`, pour éviter de *flusher* trop souvent. Dans le cadre de ce cours, ces questions de performances ne nous intéressent pas, donc nous continuerons à utiliser `std::endl`.

11.5.3. Les modificateurs de flux

Nous avons déjà vu un modificateur de flux au début de ce cours : `std::boolalpha`. Il permet d'afficher `true` ou `false` au lieu de `1` ou `0` quand on manipule des booléens. Il est tout à fait possible de l'utiliser avec d'autres types de flux. L'exemple suivant vous le démontre.

```
 1  #include <fstream>
 2
 3  int main()
 4  {
 5      std::ofstream fichier { "sortie.txt" };
 6      // On va écrire true.
 7      fichier << std::boolalpha << true << std::endl;
 8      // On revient comme avant et on va écrire 0.
 9      fichier << std::noboolalpha << false << std::endl;
10
11      return 0;
12  }
```

Le modificateur `std::noboolalpha` fait l'opération inverse de `std::boolalpha`. La plupart des modificateurs sont ainsi disponibles « en paire », l'un activant le comportement désiré, l'autre le désactivant.

Voici ce que vous allez obtenir dans le fichier de sortie.

```
1  true
2  0
```

Examinons un autre exemple. Vous savez que, si vous afficher dans la console un nombre positif, vous le verrez tel quel. Mais on peut aussi demander à afficher le signe `+`, tout comme le signe `-` est affiché pour les nombres négatifs.

```cpp
#include <iostream>

int main()
{
    std::cout << std::showpos << 42 << " " << 89 << " " << -48 <<
        std::endl;
    return 0;
}
```

```
+42 +89 -48
```

11.5.4. Même les chaînes y passent !

Maintenant, admettons que vous vouliez avoir une chaîne de caractères représentant un nombre réel. Mais, contraintes supplémentaires, vous voulez que le signe soit toujours affiché et que le nombre soit écrit en utilisant la notation scientifique . Comment faire ?

Il y a bien `std::scientific` et `std::showpos`, mais ils n'existent que pour les flux. Justement, la solution va être de **traiter les chaînes de caractères comme des flux**. Cela se fait en utilisant `std::ostringstream` comme flux de sortie et `std::istringstream` comme flux d'entrée. Les deux sont disponibles dans le fichier `<sstream>`.

```cpp
#include <iostream>
#include <sstream>
#include <string>

int main()
{
    double const reel { 10245.789 };

    std::ostringstream flux_chaine;
    // Notez comment l'utilisation est identique à ce que vous
        connaissez.
    flux_chaine << std::scientific << std::showpos << reel <<
        std::endl;
    // On récupère une chaîne de caractères en appellant str().
    std::string resultat { flux_chaine.str() };

    std::cout << "Affichage par défaut : " << reel << std::endl;
```

```
16      std::cout << "Affichage modifié : " << resultat << std::endl;
17
18      return 0;
19  }
```

```
1  Affichage par défaut : 10245.8
2  Affichage modifié : +1.024579e+04
```

Maintenant, abordons `std::istringstream`, qui permet de traiter une chaîne de caractères comme un flux, avec un exemple concret. En informatique, beaucoup de valeurs sont écrites suivant le système hexadécimal . Que ce soit l'identifiant d'une carte réseau ou le codage des couleurs sur un site web, il est courant de l'utiliser.

L'hexadécimal

Le système hexadécimal est un système de numération qui utilise **16 chiffres** au lieu des 10 que nous connaissons dans le système décimal. Les chiffres vont de 0 à 9, comme pour le décimal, puis on utilise les lettres A, B, C, D, E et F. Ainsi, des nombres comme 2A et FD sont **des nombres écrits en hexadécimal**, qui valent respectivement 42 et 253 dans notre système décimal.
Si vous souhaitez en savoir plus, lisez l'article Wikipédia , ainsi que d'autres ressources. Pour ma part, je ne vais pas rentrer plus dans les détails que ça.

Par exemple, l'orange agrume utilisé sur Zeste de Savoir est écrit **f8ad32** dans son code source. Imaginons donc que nous récupérions cette valeur. C'est une chaîne de caractères et nous voulons en déterminer le niveau de rouge, de vert et de bleu.

Il existe un modificateur de flux appelé `std::hex` qui permet de dire au flux concerné que les nombres qu'il va extraire utilisent le système hexadécimal et non décimal. En traitant une chaîne de caractères comme un flux, on résout alors notre problème.

```
1   #include <iostream>
2   #include <sstream>
3
4   int main()
5   {
6       std::istringstream flux_entree { "f8 ad 32" };
7       int rouge { 0 };
8       int vert { 0 };
9       int bleu { 0 };
10
11      flux_entree >> std::hex >> rouge >> vert >> bleu;
12      std::cout << "Niveau de rouge : " << rouge << std::endl;
13      std::cout << "Niveau de vert : " << vert << std::endl;
14      std::cout << "Niveau de bleu : " << bleu << std::endl;
```

```
15
16     return 0;
17 }
```

```
1 Niveau de rouge : 248
2 Niveau de vert : 173
3 Niveau de bleu : 50
```

Zeste d'histoire

On utilisait beaucoup `std::ostringstream` et `std::istringstream` avant C++11 pour convertir des nombres en chaînes de caractères et vice-versa. D'autres méthodes existent maintenant, mais nous en parlerons plus tard.

11.5.5. En résumé

— Pour ouvrir un fichier, il faut préciser son chemin, qui peut être relatif ou absolu.
— On écrit dans un fichier en utilisant `std::ofstream`.
— On lit dans un fichier en utilisant `std::ifstream`.
— Les flux sont une abstraction permettant d'envoyer ou recevoir un flux de données à des composants quelconques.
— Les flux sont bufferisés.
— Il existe des flux pour communiquer avec des fichiers (`std::ifstream` et `std::ofstream`), avec des chaînes de caractères (`std::istringstream` et `std::ostringstream`), avec l'entrée et la sortie standards (`std::cin` et `std::cout`).
— On utilise des modificateurs de flux pour en modifier le comportement.

Voilà, nous avons vu les bases et vous commencez déjà à faire des choses intéressantes. Mais vous ne connaissez pas tout, loin de là. Par exemple, nous ne savons pas comment découper notre programme. En effet, nous écrivons tout dans le `main`. Heureusement, la prochaine partie est là pour nous.

Contenu masqué

Contenu masqué n°29 :

Correction statistiques sur des fichiers

```cpp
#include <fstream>
#include <iostream>
#include <string>

int main()
{
    std::ifstream fichier { "fichier_1.txt" };
    std::string ligne { "" };

    int total_lignes { 0 };
    int total_caracteres { 0 };
    int total_mots { 0 };

    bool espace_caractere_precedent { false };
    while (std::getline(fichier, ligne))
    {
        ++total_lignes;

        for (char c : ligne)
        {
            if (isspace(c))
            {
                // Si le précédent n'était pas un espace, alors
                  c'était une partie de mot.
                if (!espace_caractere_precedent)
                {
                    ++total_mots;
                }

                espace_caractere_precedent = true;
            }
            else
            {
                ++total_caracteres;
                espace_caractere_precedent = false;
            }
        }

        // Ne pas oublier de regarder pour l'éventuel dernier mot.
        if (!espace_caractere_precedent)
        {
            ++total_mots;
            espace_caractere_precedent = true;
        }
    }

```

```
46      std::cout << "Total de caractères : " << total_caracteres <<
            std::endl;
47      std::cout << "Total de lignes : " << total_lignes << std::endl;
48      std::cout << "Total de mots : " << total_mots << std::endl;
49
50      return 0;
51  }
```

Félicitations, vous avez déjà codé une ébauche de wc, un programme UNIX existant. Retourner au texte.

Troisième partie

On passe la deuxième !

Depuis les débuts de l'informatique et de la programmation, découper son code en sous-morceaux est un principe de base. Celui-ci permet de mieux diviser le travail, résoudre les problèmes plus facilement, aide les développeurs à mieux s'y retrouver. En bref, **plein d'avantages**.

Cette partie va non seulement nous permettre d'apprendre toutes les notions de découpages en C++, mais également nous introduire à tout un tas de notions nouvelles qui vont grandement élargir notre champs de possibilités.

12. Découpons du code — Les fonctions

Nos programmes deviennent de plus en plus complets, nous sommes en mesure de faire de plus en plus de choses. Mais, chaque fois que nous avons besoin de réutiliser un morceau de code, comme dans le cadre de la protection des entrées, nous nous rendons comptes que nous devons **dupliquer** notre code. Autant dire que c'est **contraignant**, ça **alourdit le code** et c'est une **source d'erreurs potentielles**.

Le moment est donc venu pour vous de découvrir **les fonctions**, à la base de nombreux codes et la solution à notre problème.

12.1. Les éléments de base d'une fonction

12.1.1. Une fonction, c'est quoi ?

Une fonction est un **ensemble d'instructions** pour réaliser une tâche précise. Cet ensemble d'instructions est isolé dans une partie spécifique qui porte un nom, appelé **identificateur**. Enfin, une fonction peut ou non prendre des informations en entrée, comme elle peut ou non fournir des informations en sortie.

Depuis le début de ce cours, nous manipulons des fonctions inclues dans la bibliothèque standard. Revenons un peu sur différents exemples qui vont illustrer la définition théorique vue ci-dessus.

12.1.1.1. Cas le plus simple

Nous avons vu des fonctions simples qui ne renvoyaient ni ne prenaient de valeur. Ce genre de fonction n'a pas besoin qu'on lui fournisse des données en entrées, et elle n'en fournit pas en retour.

```cpp
1 std:string chaine { "Salut toi" };
2 // On supprime le dernier caractère.
3 chaine.pop_back();
4 // On supprime la chaîne.
5 chaine.clear();
6 
7 std::list<int> liste { 4, -9, 45, 3 };
8 // On trie la liste.
9 liste.sort();
```

12.1.1.2. Fonction renvoyant une valeur

Certaines fonctions nous fournissent des informations que nous pouvons, au choix, récupérer ou ignorer. Le chapitre sur les tableaux nous en fournit plusieurs exemples.

```
std::vector<int> const tableau_de_int { 1, 2, 3, 4, 5, 6, 7, 8, 9
    };
// Récupérer le premier élément.
int premier { tableau_de_int.front() };
// Récupérer le dernier élément.
int dernier { tableau_de_int.back() };
```

Nous ne sommes pas obligés de stocker la valeur qu'une fonction renvoie. Par exemple, `std::getline` peut s'utiliser tel quel, sans se soucier de ce que la fonction a renvoyé.

12.1.1.3. Fonction attendant une valeur

D'autres fois, certaines fonctions ont besoin d'informations extérieures pour travailler. C'est le cas notamment des algorithmes que nous avons vus, mais aussi d'autres fonctions.

```
std::string texte { "Du texte." };
// Ajouter une lettre à la fin.
texte.push_back('!');

std::vector<char> tableau {};
// On remplit le tableau avec cinq fois la valeur 'C'.
tableau.assign(5, 'C');
```

> **Point de vocabulaire**
>
> Dans le cas où une fonction ne retourne aucune valeur, on parle dans la littérature informatique de **procédure**. Cette appellation est valable, peu importe si la fonction attend ou non des données.

12.1.1.4. Fusion !

Enfin, dernière possibilité, il y a les fonctions qui attendent des informations en entrée et qui en fournissent en sortie. Là encore, nous en avons vu dans le cours.

```
std::vector<int> tableau { 1, 2, 3, 4 };
// Renvoie un itérateur sur le premier élément.
auto it = std::begin(tableau);
```

```
4  // On compte le nombre d'éléments valant deux.
5  std::count(std::begin(tableau), std::end(tableau), 2);
6
7  std::string texte { "" };
8  // std::getline attend qu'on lui donne le flux à lire et la chaîne
       qui servira à stocker le résultat.
9  // Elle renvoie ce flux en gise de retour.
10 std::getline(std::cin, texte);
11 // Attend un conteneur en entrée et renvoie sa taille en sortie.
12 auto taille { std::size(texte) };
```

12.1.2. Une fonction bien connue

Nous avons un exemple de fonction sous nos yeux depuis le premier code que vous avez écrit : la fonction `main`. Celle-ci est le point d'entrée de tous les programmes que nous codons. Je la remets ci-dessous.

```
1 int main()
2 {
3     // Instructions diverses.
4     return 0;
5 }
```

- D'abord, `main` est le nom de la fonction, son **identificateur**.
- Ensuite, le `int` situé juste avant définit le **type de retour**. Dans notre cas, la fonction renvoie une valeur entière, qui est le `0` que nous voyons dans `return 0;` et qui indique au système d'exploitation que tout s'est bien passé.
- Après, nous voyons une paire de parenthèses vides, ce qui signifie que la fonction n'attend **aucune information en entrée**.
- Enfin, entre les accolades `{}`, nous avons le **corps de la fonction**, les **instructions** qui la composent.

12.1.3. Les composants d'une fonction

13. Schéma d’une fonction

En écrivant en pseudo-code, voici à quoi ressemble une fonction en C++.

```
type_de_retour identificateur(paramètres)
{
    instructions
}
```

13.0.0.1. Votre identité, s’il vous plaît

Le point le plus important est de donner un nom à notre fonction. Les règles sont les mêmes que pour nommer nos variables. Je les réécris ci-dessous.

- L’identificateur doit **commencer par une lettre**. Il ne peut pas commencer par un chiffre, c’est interdit. Il ne doit pas commencer non plus par *underscore* `_`, car les fonctions commençant par ce caractère obéissent à des règles spéciales et sont souvent réservées à l’usage du compilateur.
- Les espaces et les signes de ponctuation sont **interdits** (`'`, `?`, etc).
- **On ne peut pas utiliser un mot-clef** du langage comme identificateur. Ainsi, il est interdit de déclarer une fonction s’appelant `int` ou `return`, par exemple.

Je vous rappelle également l’importance de donner un nom clair à vos fonctions, qui définit clairement ce qu’elles font. Évitez donc les noms trop vagues, les abréviations obscures, etc.

13.0.0.2. Mais de quel type es-tu ?

Comme vous le savez, une fonction peut ou non renvoyer une valeur de retour, comme nous l’avons vu à l’instant. Si elle doit renvoyer une valeur quelconque, alors il suffit d’écrire le type de cette valeur.

À l’inverse, si votre fonction ne doit rien renvoyer, alors **on utilise le mot-clef `void`**, qui signifie « vide » et qui, dans ce contexte, indique que la fonction renvoie du vide, c’est-à-dire rien.

Pour renvoier une valeur, on utilise le mot-clef `return`, exactement comme dans la fonction `main`. Il peut y en avoir **plusieurs** dans une fonction, par exemple deux `return`, un si un `if` est vrai, l’autre s’il est faux. Par contre, lorsqu’une instruction `return` est exécutée, **on sort de la fonction en cours** et tout le code restant n’est pas exécuté. On dit qu’un retour de fonction **coupe le flot d’exécution de la fonction**. Oui, comme `break` dans une boucle. Vous suivez, c’est très bien.

13.0.0.3. Paramétrage en cours...

Maintenant, parlons des paramètres. Grâce aux paramètres, une fonction montre au reste du monde **ce qu'elle attend pour travailler**. Par exemple, `std::sort` attend qu'on lui donne un itérateur de début et un itérateur de fin. Quand nous ajoutons un élément à un tableau à l'aide de `push_back`, il faut indiquer à cette dernière la valeur que nous ajoutons.

Comment les déclarer ? C'est très simple, il faut que chaque paramètre ait **un type** et **un identificateur**. S'il y en a plusieurs, ils seront **séparés par des virgules**.

13.0.0.4. Exemples

Voici un exemple de fonctions que j'ai créé en reprenant les codes que nous avions écrits il y a plusieurs chapitres.

```
int pgcd(int a, int b)
{
    int r { a % b };
    while (r != 0)
    {
        a = b;
        b = r;
        r = a % b;
    }

    // On peut tout à fait renvoyer la valeur d'un paramètre.
    return b;
}

int somme(int n)
{
    // On peut renvoyer le résultat d'un calcul ou d'une expression
        directement.
    return (n * (n + 1)) / 2;
}

bool ou_exclusif(bool a, bool b)
{
    return (a && !b) || (b && !a);
}

int main()
{
    int const a { 845 };
    int const b { 314 };

    std::cout << "Le PGCD de " << a << " et " << b << " vaut " <<
        pgcd(a, b) << "." << std::endl;
```

```
32     std::cout << "La somme de tous les entiers de 1 à 25 vaut " <<
           somme(25) << "." << std::endl;
33
34     std::cout << std::boolalpha;
35     std::cout << "XOR(true, false) vaut " << ou_exclusif(true,
           false) << "." << std::endl;
36
37     return 0;
38 }
```

Notez qu'on peut très bien donner en argument à une fonction une variable qui porte le même identificateur qu'un des paramètres. Il n'y a aucun problème à ça, car nous sommes dans **deux portées différentes**: `main` et la fonction en question.

Conséquence du point précédent, il n'est **pas possible d'utiliser un paramètre de fonction en dehors de celle-ci**. Le code suivant produit donc une erreur.

```
 1 void fonction(int parametre)
 2 {
 3     // Aucun problème.
 4     parametre = 5;
 5 }
 6
 7 int main()
 8 {
 9     // La variable parametre n'existe pas !
10     parametre = 410;
11     return 0;
12 }
```

i Argument et paramètre

Chez beaucoup de personnes, la distinction entre paramètre et argument n'est pas claire.

— Un paramètre, aussi appelé paramètre formel, c'est **ce qu'attend une fonction pour travailler** et qui est inscrit dans sa déclaration.
— Un argument, aussi appelé paramètre réel, c'est **la valeur transmise à la fonction quand on l'utilise**.

```
void fonction(int entier)
{
    // entier est le paramètre de la fonction.
}

int main()
{
    // 4 est l'argument de la fonction.
    fonction(4);

    int const valeur { 5 };
    // valeur est l'argument de la fonction.
    fonction(valeur);
    return 0;
}
```

En pratique, beaucoup utilisent les deux de façon interchangeable et vous serez parfaitement compris si vous faites de même. Je tenais néanmoins à vous expliquer cela rigoureusement.

13.1. Exercices

Il est temps de pratiquer toutes ces nouvelles notions que vous avez acquises, afin de mieux les comprendre et bien les retenir.

13.1.1. Afficher un rectangle

Commençons par une fonction simple qui **affichera un rectangle** composé d'étoiles `*`. Voici un exemple en lui donnant une longueur de 4 et une largeur de 3.

```
***
***
***
***
```

Correction affichage de rectangles

13.1.2. Distributeur d'argent

Le but est de **donner une quantité de coupures correspondant à une somme donnée**, en utilisant les plus grosses en premiers. Par exemple, 285€ avec des coupures de 500€, 200€, 100€, 50€, 20€, 10€, 5€, 2€ et 1€ donne un total d'un billet de 200€, un billet de 50€, un billet de 20€, un billet de 10€ et enfin, un billet de 5€.

Par contre, 346€ avec uniquement 20€, 10€, 5€, 2€ et 1€ donne dix-sept billets de 20€, un billet de 5€ et une pièce de 1€. Il faut donc prendre en compte les coupures disponibles dans le résultat.

Correction distributeur d'argent

Bonus : faire une fonction qui affiche de façon lisible le total. Par exemple, pour 285€, on peut imaginer la sortie suivante.

```
1 x 200 euros
1 x 50 euros
1 x 20 euros
1 x 10 euros
1 x 5 euros
```

Correction bonus

13.1.3. Parenthésage

Imaginons une expression de plusieurs dizaines de caractères, avec des parenthèses ouvrantes `(` et fermantes `)`. Il serait fastidieux de regarder manuellement si une expression est correctement parenthésée, c'est-à-dire qu'elle a le **même nombre de symboles ouvrants et fermants** et qu'ils sont bien **dans l'ordre**.

Ainsi, l'expression `((()))` est correcte, alors que `((())` est incorrecte parce qu'il manque une parenthèse fermante. Quant à `)(`, elle est également incorrecte parce que le symbole fermant est avant le symbole ouvrant.

Correction parenthésage

13.2. Quelles sont vos références ?

13.2.1. La problématique

J'ai écrit dans la première section que les paramètres d'une fonction étaient dans une portée différente, ce qui autorise de faire ceci.

```
void fonction(int a)
{
    a = 5;
}

int main()
{
    int a { 8 };
    // Aucun problème.
    fonction(a);

    return 0;
}
```

Il faut cependant savoir que ceci implique **une copie**. Pour chaque paramètre, le compilateur va réserver un espace en mémoire de la taille nécessaire et copier la variable qu'on lui passe. Nous allons donc manier des variables totalement différentes. C'est ce qui s'appelle le **passage par copie**.

```
#include <iostream>
#include <vector>

void fonction(int a, double b, std::vector<int> c)
{
    a = 5;
    b = 2.18781;
    c.assign(7, 0);

    // Ici, nous manipulons donc des copies.
    std::cout << a << std::endl;
    std::cout << b << std::endl;
    for (auto element : c)
    {
        std::cout << element << " ";
    }
    std::cout << std::endl << std::endl;
}

int main()
```

```
21 {
22     int const entier { 42 };
23     double const reel { 3.14159 };
24     std::vector<int> const tableau { 1, 782, 4, 5, 71 };
25 
26     // Chaque variable va être copiée.
27     fonction(entier, reel, tableau);
28 
29     // Affichons pour prouver que les variables originales n'ont
           pas changé.
30     std::cout << entier << std::endl;
31     std::cout << reel << std::endl;
32     for (auto element : tableau)
33     {
34         std::cout << element << " ";
35     }
36 
37     std::cout << std::endl;
38     return 0;
39 }
```

Le côté négatif, c'est qu'**on doit forcément copier l'argument**, même si on ne le modifie pas. Dans le cas d'un entier ou d'un caractère, c'est idéal, mais dans le cas d'un tableau de plusieurs milliers d'éléments, on perd du temps inutilement. C'est comme si je demandais à consulter un certain original dans des archives et qu'on me faisait des photocopies à la place. S'il s'agit d'un post-it, pas grave, mais s'il s'agit d'un livre de 400 pages, bonjour le gâchis !

13.2.2. Les références

La solution consiste à ne pas passer une copie de la variable originale mais **un alias**, **une référence** vers celle-ci et qui soit manipulable comme si j'avais l'original entre les mains. Cela revient à me donner le document original et éventuellement, selon le contexte, le droit de le modifier.

Une référence en C++ est simple à déclarer mais doit réunir plusieurs conditions.

— On utilise l'esperluette `&` pour ça.
— Elle doit être **de même type que la variable cible**. Une référence sur un `double` ne peut pas se voir associée à une variable de type `int`.
— Une référence **cible une seule et unique variable**. On ne peut pas créer de référence qui ne cible rien, ni changer la cible d'une référence une fois qu'on l'a créée.
— Une référence peut être **définie comme constante**, ce qui implique qu'on ne pourra pas modifier sa valeur.

```
1 #include <iostream>
2 
3 int main()
```

```cpp
{
    int entier { 40 };
    // On initialise notre référence pour qu'elle référence entier.
    int & reference_entier { entier };

    std::cout << "Entier vaut : " << entier << std::endl;
    std::cout << "Référence vaut : " << reference_entier <<
        std::endl;

    reference_entier += 2;
    std::cout << "Entier vaut : " << entier << std::endl;
    std::cout << "Référence vaut : " << reference_entier <<
        std::endl;

    int const entier_constant { 0 };
    // On peut également déclarer des références constantes sur des
        entiers constants.
    int const & reference_const_entier_constant { entier_constant
        };
    // Ce qui interdit ce qui suit, car le type référencé est un
        int const.
    // reference_const_entier_constant = 1;

    // On ne peut pas déclarer de référence non constante sur un
        entier constant.
    //int & reference_entier_constant { entier_constant };

    // On peut par contre tout à fait déclarer une référence
        constante sur un objet non constant.
    // On ne peut donc pas modifier la valeur de l'objet en
        utilisant la référence, mais l'entier
    // original reste tout à fait modifiable.
    int const & reference_const_entier { entier };
    std::cout << "Entier vaut : " << entier << std::endl;
    std::cout << "Référence constante vaut : " <<
        reference_const_entier << std::endl;

    entier = 8;
    std::cout << "Entier vaut : " << entier << std::endl;
    std::cout << "Référence constante vaut : " <<
        reference_const_entier << std::endl;

    return 0;
}
```

Tableaux et références

Vous avez déjà utilisé des références sans le savoir. En effet, quand, dans le chapitre précédent, nous accédons à un élément d'un tableau, c'est en fait une référence qui est renvoyée, ce qui autorise à modifier un tableau comme nous l'avions vu.

```
// On change la valeur du premier élément.
tableau[0] = 5;
```

C++ est un langage qui cache plein de concepts sous des airs innocents.

13.2.3. Paramètres de fonctions

On peut tout à fait déclarer les paramètres d'une fonction comme étant des références. Ainsi, celles-ci ne recevront plus une copie mais bel et bien des références sur les objets. On parle donc de **passage par référence**. On peut, bien entendu, préciser si la référence est constante ou non. Dans le premier cas, toute modification du paramètre sera interdite, alors que le deuxième cas autorise à le modifier et conserver ces changements.

```
#include <iostream>
#include <string>

void fonction_reference_const(std::string const & texte)
{
    std::cout <<
        "J'ai reçu une référence comme paramètre. Pas de copie, youpi !"
        << std::endl;
    // Interdit.
    // texte = "Changement";
}

void fonction_reference(std::string & modifiable)
{
    std::cout <<
        "Je peux modifier la chaîne de caractères et les changements seront co
        << std::endl;
    modifiable = "Exemple";
}

int main()
{
    std::string modifiable { "Du texte modifiable." };
    std::string const non_modifiable {
        "Du texte qui ne sera pas modifiable." };

```

```
22      std::cout << "Avant : " << modifiable << std::endl;
23      fonction_reference(modifiable);
24      std::cout << "Après : " << modifiable << std::endl;
25
26      fonction_reference_const(non_modifiable);
27      // Possible.
28      fonction_reference_const(modifiable);
29      // Impossible, parce que la fonction attend une référence sur
            un std::string et nous lui donnons une référence sur un
            std::string const.
30      //fonction_reference(non_modifiable);
31
32      return 0;
33  }
```

? Quand dois-je utiliser une référence constante ou une référence simple ?

Il suffit de se poser la question : « est-ce que ce paramètre est destiné à être modifié ou non ? » Si c'est le cas, alors on utilisera une référence simple. Dans le cas contraire, il faut utiliser une référence constante. Ainsi, l'utilisateur de la fonction aura des garanties que son objet ne sera pas modifié dans son dos.

13.2.3.1. Le cas des littéraux

Analysez le code suivant, sans le lancer, et dites-moi ce que vous pensez qu'il va faire. Si vous me dites qu'il va simplement afficher deux messages… dommage, vous avez perdu.

```
1   #include <iostream>
2
3   void test(int & a)
4   {
5       std::cout << "Je suis dans la fonction test.\n";
6   }
7
8   void test_const(int const & a)
9   {
10      std::cout << "Je suis dans la fonction test_const.\n";
11  }
12
13  int main()
14  {
15      test(42);
16      test_const(42);
17      return 0;
18  }
```

En fait, ce code ne va même pas compiler, à cause de la ligne 15. Il y a une très bonne raison à ça : nous passons un littéral, qui est **invariable** et **n'existe nulle part en mémoire**, or nous tentons de le donner en argument à une fonction qui attend une référence, un alias, sur une variable modifiable. Mais un littéral n'est pas modifiable et tenter de le faire **n'a aucun sens**. Le compilateur refuse donc et ce code est invalide.

Par contre, supprimez-la et le code compilera. Pourtant, la seule différence entre les deux fonctions, c'est la présence d'un `const`. *Eh* bien c'est lui qui rend le code valide. En effet, le compilateur va **créer un objet temporaire anonyme**, qui n'existera que le temps que la fonction prendra pour travailler, et va lui assigner le littéral comme valeur. On peut réécrire le code précédent comme ceci pour mieux comprendre.

```
#include <iostream>
#include <string>

void test_const(int const & a)
{
    std::cout << "Je suis dans la fonction test_const.\n";
}

void test_string_const(std::string const & texte)
{
    std::cout << "Je suis dans la fonction test_string_const.\n";
}

int main()
{
    // test_const(42);
    {
        // Nouvelle portée.
        int const variable_temporaire { 42 };
        int const & reference_temporaire { variable_temporaire };
        test_const(reference_temporaire);
        // Nos variables disparaissent.
    }

    // Exemple avec une chaîne de caractères.
    // test_string_const("Salut toi !");
    {
        std::string const texte_temporaire { "Salut toi !" };
        std::string const & reference_temporaire { texte_temporaire
            };
        test_string_const(reference_temporaire);
    }

    return 0;
}
```

13.2.3.2. Le cas des types standard

Utiliser des références pour `std::vector` ou `std::string` d'accord, mais qu'en est-il des types simples comme `int`, `char` et `double` ? Y a-t-il un intérêt à utiliser des références constantes ? Doit-on bannir toutes les copies de nos codes sous prétexte d'efficacité ? Non.

L'usage en C++, c'est que les types natifs sont **toujours passés par copie**. Pourquoi cela ? Parce qu'ils sont très petits et le coût de la création d'une référence sur des types aussi simples est souvent plus élevé que celui d'une bête copie. En effet, le compilateur peut **optimiser la copie** et la rendre extrêmement rapide, bien plus qu'avec les références.

Donc ne tombez pas dans le piège de **l'optimisation prématurée**, qui consiste à vouloir améliorer les performances de son programme en faisant des changements qui n'apporteront aucune amélioration. Laissez plutôt ce travail au compilateur.

13.2.4. Valeur de retour

Les références ont une contrainte particulière : **elles doivent toujours être valides**. Dans le cas d'un paramètre de fonction, il n'y a aucun danger. En effet, l'argument transmis à la fonction existera toujours quand celle-ci se terminera, comme le montre le code suivant.

```
void fonction(int & reference)
{
    // On modifie une référence sur une variable qui existe, pas de
        soucis.
    reference = 42;
}

int main()
{
    int variable { 24 };
    // On transmet variable...
    fonction(variable);
    // ... qui existe toujours après la fin de la fonction.

    return 0;
}
```

Cependant, dans le cas où l'on souhaite qu'une fonction renvoie une référence, il faut faire très attention parce qu'on peut **renvoyer une référence sur un objet qui n'existe plus** ! Ceci va donc entraîner un **comportement indéterminé**.

```
int& fonction()
{
    int variable { 42 };
```

```
4	    // Oho, variable sera détruite quand la fonction sera terminée
	        !
5	    return variable;
6	}
7	
8	int main()
9	{
10	    // Grosse erreur, notre référence est invalide !
11	    int& reference { fonction() };
12	
13	    return 0;
14	}
```

Référence locale

Ne renvoyez JAMAIS de référence sur une variable locale.

13.2.5. Un mot sur la déduction de type

Certains reprochent **son côté « rigide » à ce mot-clé**. En effet, `auto` seul dégage la référence. À l'inverse, `auto &` sera toujours une référence. Et il en est de même pour `const` : `auto const` sera toujours constant et `auto const &` sera toujours une référence sur un objet constant.

```
1	int main()
2	{
3	    int variable { 2 };
4	    int & reference { variable };
5	
6	    // Sera toujours une variable et jamais une référence.
7	    auto toujours_variable_1 { variable };
8	    auto toujours_variable_2 { reference };
9	
10	    // Sera toujours une référence et jamais une variable.
11	    auto & toujours_reference_1 { variable };
12	    auto & toujours_reference_2 { reference };
13	
14	    // On pourrait rajouter des exemples avec const, mais vous
	        comprenez le principe.
15	    return 0;
16	}
```

Il existe cependant un mot-clef qui s'adapte toujours au type en paramètre parce qu'il conserve la présence (ou non) de `const` et de références. Il s'appelle `decltype` et s'utilise de deux façons. La première, c'est avec une **expression explicite**, comme un calcul, une autre variable ou le

retour d'une fonction, ce qui donne `decltype(expression) identifiant { valeur };` et est illustré ci-dessous.

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> tableau { -8, 45, 35, 9 };

    // Sera toujours une variable.
    auto premier_element_variable { tableau[0] };
    // Est en fait une référence sur cet élément, ce qui autorise à
        le modifier.
    decltype(tableau[1]) deuxieme_element_reference { tableau[1] };

    // Modifions et regardons.
    premier_element_variable = 89;
    deuxieme_element_reference = 0;

    for (auto const & element : tableau)
    {
        std::cout << element << std::endl;
    }

    return 0;
}
```

La deuxième façon de l'utiliser, c'est de combiner `decltype` et `auto`, donnant `decltype(auto) identifiant { valeur };` afin d'avoir le type exact de `valeur` et de ne pas se répéter. La ligne 11 de notre code précédent peut ainsi être raccourcie.

```cpp
std::vector<int> tableau { -8, 45, 35, 9 };
// Sera de type int& si tableau est modifiable, de type int const &
    si tableau est déclaré comme const.
decltype(auto) deuxieme_element_reference { tableau[1] };
```

Ces deux mots-clefs ne sont pas deux concurrents s'opposant pour faire le même travail, bien au contraire. Ils sont une preuve de la liberté que C++ laisse au développeur. Nous verrons plus tard dans ce cours d'autres usages, qui rendront ces mots-clefs encore plus concrets pour vous. En attendant, retenez ceci pour vous aider à choisir.

- Choisissez `auto` lorsque vous souhaitez explicitement une valeur (par copie).
- Choisissez `auto &` (`const` ou non) lorsque vous souhaitez explicitement une référence.
- Choisissez `decltype` lorsque vous souhaitez le type exact.

13.3. Nos fonctions sont surchargées !

Dans le chapitre sur les fichiers, je vous ai appris qu'il y a, en plus de `std::ostringstream` et `std::istringstream`, d'autres méthodes pour faire la conversion nombre <-> chaîne de caractères.

Voici, ci-dessous, comment cette fonction est déclarée dans la bibliothèque standard.

```
std::string to_string(int value);
std::string to_string(double value);
```

? Attends une seconde ! Comment c'est possible qu'on ait deux fonctions avec le même nom ?

La réponse est simple : ce qui différencie deux fonctions, ça n'est pas seulement leur identificateur mais également leurs paramètres. C'est ce qu'on appelle la **signature d'une fonction**. Dans notre cas, le compilateur voit deux signatures différentes : `to_string(int)` et `to_string(double)`, sans que ça ne pose le moindre problème.

Cette possibilité s'appelle **surcharge**, de l'anglais « *overloading* ». Elle rend le code plus clair car on laisse au compilateur le soin d'appeler la bonne surcharge.

```
#include <iostream>
#include <string>

int main()
{
    int const entier { -5 };
    double const flottant { 3.14156 };

    std::string affichage { "Voici des nombres : " };
    affichage += std::to_string(entier) + ", ";
    affichage += std::to_string(flottant);

    std::cout << affichage << std::endl;

    return 0;
}
```

Par contre, **le type de retour n'entre pas en compte dans la signature** d'une fonction. Ainsi, deux fonctions avec le même identifiant, les mêmes paramètres et des types de retour différents seront rejetées par le compilateur car il ne saura pas laquelle utiliser. Et c'est logique, car comment, en lisant le code, sauriez-vous ce que la fonction va vous renvoyer ?

C'est pour ça que les fonctions convertissant des chaînes de caractères en nombres ont des noms bien distincts : `std::stoi` pour convertir une chaîne en `int` et `std::stod` pour `double`.

```cpp
#include <iostream>
#include <string>

int main()
{
    int const entier { std::stoi("-8") };
    double const flottant { std::stod("2.71878") };

    std::cout << entier + flottant << std::endl;
    return 0;
}
```

13.4. [T.P] Gérer les erreurs d’entrée — Partie III

Avec ce que nous avons vu, nous sommes maintenant capable de faire des fonctions qui vont nous épargner d’avoir à dupliquer plusieurs fois le même code chaque fois que nous souhaiterons sécuriser nos entrées. Nous pourrions ainsi sécuriser le code suivant sans duplication.

```cpp
#include <iostream>

int main()
{
    int jour { 0 };
    std::cout << "Quel jour es-tu né ? ";
    // Entrée.

    int mois { 0 };
    std::cout << "Quel mois ? ";
    // Entrée.

    int annee { 0 };
    std::cout << "Quelle année ? ";
    // Entrée.

    double taille { 0.0 };
    std::cout << "Quelle taille ? ";
    // Entree.

    std::cout << "Tu es né le " << jour << "/" << mois << "/" <<
        annee << " et tu mesures " << taille << "m." << std::endl;
    return 0;
}
```

Correction T.P Partie III

13.5. Dessine-moi une fonction

Nous allons nous attarder sur un petit problème qui peut apparaître lorsque l'on crée des fonctions : les appels croisés.

13.5.1. Le problème

Introduisons ce problème par un exemple : essayons d'implémenter deux fonctions `est_pair` et `est_impair`, dont l'objectif est, comme leurs noms l'indiquent, de dire si un entier est pair ou impair. Une manière élégante de procéder est de faire en sorte que chacune appelle l'autre.

```
bool est_pair(int i)
{
    if (i == 0)
    {
        return true;
    }
    else if (i < 0)
    {
        return est_pair(-i);
    }
    else
    {
        return est_impair(i - 1);
    }
}

bool est_impair(int i)
{
    if (i < 0)
    {
        return est_impair(-i);
    }
    else
    {
        return est_pair(i - 1);
    }
}
```

L'idée est de s'approcher pas à pas du cas `0` (dont on connaît la parité) en appelant la fonction avec l'entier `i-1`. C'est ce qu'on appelle de la programmation récursive, rappelant l'idée d'une récurrence en mathématiques.

Bref, ce qui est important ici, c'est que ce code ne compile pas. En effet, comme le compilateur lit le code de haut en bas, il ne connaît pas la fonction `est_impair` lorsqu'on l'utilise dans `est_pair`. Et ça ne marche pas non plus en échangeant les deux fonctions, puisque `est_impair` appelle aussi `est_pair`.

Heureusement, nos fonctions ne sont pas à jeter puisque C++ est bien fait, et qu'il a donc une solution à ce problème : **les prototypes**.

13.5.2. La solution

Un prototype est une expression qui décrit toutes les informations qui définissent une fonction, à part son implémentation.

— Son identificateur.
— Son type de retour.
— Ses paramètres.

On peut résumer en disant que c'est tout simplement la première ligne de la définition d'une fonction, à laquelle on ajoute un point-virgule.

```
type_de_retour identifiant(paramètres);
```

Ces informations sont suffisantes pour que le compilateur sache que la fonction existe et qu'il sache comment elle s'utilise. Cela nous permet alors d'écrire, pour notre exemple précédent :

```
bool est_impair(int i);

bool est_pair(int i)
{
    if (i == 0)
    {
        return true;
    }
    else if (i < 0)
    {
        return est_pair(-i);
    }
    else
    {
        return est_impair(i - 1);
    }
}

bool est_impair(int i)
{
    if (i < 0)
    {
```

```
        return est_impair(-i);
    }
    else
    {
        return est_pair(i - 1);
    }
}
```

Et maintenant, ça compile !

13.5.3. Utilisons les prototypes

Les prototypes ne servent pas qu'à résoudre les problèmes d'appels croisés. En effet, ils peuvent être utilisés pour améliorer la lisibilité du code. Tout comme le compilateur, **nous n'avons pas besoin de l'implémentation pour savoir comment s'utilise une fonction**. En plus, le nom nous suffit souvent, lorsqu'il est bien choisi, pour savoir **à quoi sert la fonction**.

Constat : **nous ne voulons pas voir l'implémentation**. Derrière cette phrase un tantinet abusive, je veux dire que l'essentiel de l'information est réunie dans le prototype.

On peut donc tirer parti de cela pour rendre nos programmes plus lisibles. Par exemple, on peut mettre tous les prototypes avant la fonction `main` et reléguer les implémentations à la fin du fichier, juste après cette dernière. Ainsi, lorsque l'on veut utiliser une fonction, le prototype n'est plus noyé dans les implémentations et on peut en un coup d'œil récupérer toutes les informations qui nous intéressent.

Cette idée de **faire valoir les interfaces plutôt que les implémentations** est très importante en programmation et l'on aura l'occasion de la recroiser plusieurs fois.

13.5.4. En résumé

— Une fonction permet de factoriser un morceau de code utilisé régulièrement. Nous en utilisons beaucoup depuis le début de ce cours.
— Une fonction a un identifiant et peut prendre ou non plusieurs paramètres, ainsi que renvoyer ou non une valeur. Dans le cas où elle ne renvoie rien, elle est déclarée avec le mot-clef `void`.
— Les références sont des alias de variables qui permettent, à travers un autre nom, de les manipuler directement. On s'en sert, notamment, dans les paramètres des fonctions pour éviter des copies inutiles ou pour conserver les modifications faites sur un objet.
— Il faut faire attention à ce qu'une référence soit toujours valide. Cela nous interdit donc de renvoyer une référence sur une variable locale.
— Le mot-clef `auto` ne conserve pas les références, mais le nouveau mot-clef que nous avons introduit, `decltype`, si.
— La signature d'une fonction est ce qui la rend unique et est composée de son identifiant et de ses paramètres. On peut donc avoir plusieurs fonctions avec le même identifiant si les paramètres sont différents, ce qu'on appelle la surcharge.

— Le prototype d'une fonction permet, entre autres, de résoudre le problème des appels croisés.
— Le prototype contenant l'identificateur, le type de retour et les paramètres d'une fonction, il donne suffisamment d'informations pour ne pas avoir besoin de lire l'implémentation.

Contenu masqué

Contenu masqué n°30 : Correction affichage de rectangles

```
#include <iostream>

void rectangle(int longueur, int largeur)
{
    for (auto i { 0 }; i < longueur; ++i)
    {
        for (auto j { 0 }; j < largeur; ++j)
        {
            std::cout << "*";
        }

        std::cout << std::endl;
    }
}

int main()
{
    rectangle(2, 5);
    rectangle(4, 3);
    return 0;
}
```

Retourner au texte.

Contenu masqué n°31 : Correction distributeur d'argent

```
#include <iostream>
#include <vector>

std::vector<int> distributeur(int total, std::vector<int>
    coupures_disponibles)
```

```
5  {
6      std::vector<int> resultat {};
7
8      for (auto coupure : coupures_disponibles)
9      {
10         resultat.push_back(total / coupure);
11         total %= coupure;
12     }
13
14     return resultat;
15 }
16
17 int main()
18 {
19     std::vector<int> const coupures_disponibles { 500, 200, 100,
           50, 20, 10, 5, 2, 1 };
20     auto const coupures { distributeur(285, coupures_disponibles)
           };
21
22     for (auto coupure : coupures)
23     {
24         std::cout << coupure << std::endl;
25     }
26
27     return 0;
28 }
```

Retourner au texte.

Contenu masqué n°32 : Correction bonus

```
1  #include <iostream>
2  #include <vector>
3
4  std::vector<int> distributeur(int total, std::vector<int>
       coupures_disponibles)
5  {
6      std::vector<int> resultat {};
7
8      for (auto coupure : coupures_disponibles)
9      {
10         resultat.push_back(total / coupure);
11         total %= coupure;
12     }
13
```

```
14      return resultat;
15  }
16
17  void affichage_distributeur(std::vector<int> total,
        std::vector<int> coupures_disponibles)
18  {
19      int i { 0 };
20      for (auto coupure : total)
21      {
22          // Si 0, ça veut dire qu'il n'y a aucune coupure de ce type
                dans le total, donc nous n'afficherons rien.
23          if (coupure != 0)
24          {
25              std::cout << coupure << " x " <<
                    coupures_disponibles[i] << " euros" << std::endl;
26          }
27          ++i;
28      }
29  }
30
31  int main()
32  {
33      std::vector<int> const coupures_disponibles { 500, 200, 100,
            50, 20, 10, 5, 2, 1 };
34      auto const coupures_285 { distributeur(285,
            coupures_disponibles) };
35      affichage_distributeur(coupures_285, coupures_disponibles);
36
37      // Moins de coupures disponibles.
38      std::vector<int> const autre_coupures_disponibles { 200, 50,
            10, 1 };
39      auto const coupures_45874 { distributeur(45874,
            autre_coupures_disponibles) };
40      affichage_distributeur(coupures_45874,
            autre_coupures_disponibles);
41
42      return 0;
43  }
```

Retourner au texte.

Contenu masqué n°33 :

Correction parenthésage

```
#include <iostream>

bool parentheses(std::string expression)
{
    int ouvrantes { 0 };
    int fermantes { 0 };

    for (auto caractere : expression)
    {
        if (caractere == '(')
        {
            ++ouvrantes;
        }
        else
        {
            ++fermantes;
        }

        if (fermantes > ouvrantes)
        {
            // Pas la peine de continuer, l'expression est
              invalide.
            return false;
        }
    }

    return ouvrantes == fermantes;
}

int main()
{
    std::cout << std::boolalpha;
    std::cout << parentheses("((()))") << std::endl;
    std::cout << parentheses("((())") << std::endl;
    std::cout << parentheses("))((") << std::endl;

    return 0;
}
```

Retourner au texte.

Contenu masqué n°34 :

Correction T.P Partie III

```cpp
#include <iostream>
#include <string>

void entree_securisee(int & variable)
{
    while (!(std::cin >> variable))
    {
        std::cout << "Entrée invalide. Recommence." << std::endl;
        std::cin.clear();
        std::cin.ignore(255, '\n');
    }
}

void entree_securisee(double & variable)
{
    while (!(std::cin >> variable))
    {
        std::cout << "Entrée invalide. Recommence." << std::endl;
        std::cin.clear();
        std::cin.ignore(255, '\n');
    }
}

int main()
{
    int jour { 0 };
    std::cout << "Quel jour es-tu né ? ";
    entree_securisee(jour);

    int mois { 0 };
    std::cout << "Quel mois ? ";
    entree_securisee(mois);

    int annee { 0 };
    std::cout << "Quelle année ? ";
    entree_securisee(annee);

    double taille { 0.0 };
    std::cout << "Quelle taille ? ";
    entree_securisee(taille);

    std::cout << "Tu es né le " << jour << "/" << mois << "/" <<
        annee << " et tu mesures " << taille << "m." << std::endl;
    return 0;
}
```

On remarque quand même que si le code n'est plus répété à chaque entrée, **nos fonctions**

sont quasiment identiques, la seule différence résidant dans le paramètre qu'elles attendent. Finalement, on duplique toujours le code. Mais comme vous vous en doutez, il y a une solution pour ça, que nous verrons en temps voulu. Retourner au texte.

14. Erreur, erreur, erreur...

Depuis le début de ce cours, nous avons été épargnés par les erreurs. Nous avons certes appris à sécuriser un peu les entrées, mais c'est tout. Ça ne sera pas toujours le cas. Il serait de bon ton que nos programmes soient **résistants aux erreurs**, afin de les rendre fiables et robustes.

Le but de ce chapitre va être d'introduire la **gestion des erreurs** ainsi que la **réflexion à mener** lors de l'écriture de son code.

14.1. L'utilisateur est un idiot

Derrière ce titre un poil provoquant se cache une vérité informatique universelle : peu importe que ce soit consciemment ou non, il arrivera **forcément** que l'utilisateur rentre une information incomplète, fausse, ou totalement différente de ce qu'on attendait. Que ce soit parce qu'il veut s'amuser avec votre programme ou bien parce qu'il a mal compris une consigne, a oublié un caractère ou s'est trompé de ligne, votre programme doit être **capable de gérer ça**.

Vérifier les entrées veut bien sûr dire s'assurer qu'elles soient du bon type, mais aussi contrôler que la donnée est cohérente et valide. Ainsi, vous ne pouvez pas dire que vous êtes nés un 38 avril, car un mois a 31 jours au maximum. Si la donnée n'est pas correcte, même si elle est du bon type, il faut demander à l'utilisateur de la ressaisir. Cela se fait relativement facilement.

```
1  #include <iostream>
2
3  int main()
4  {
5      int jour { 0 };
6      std::cout << "Donne un jour entre 1 et 31 : ";
7
8      while (!(std::cin >> jour) || jour < 0 || jour > 31)
9      {
10         std::cout << "Entrée invalide. Recommence." << std::endl;
11         std::cin.clear();
12         std::cin.ignore(255, '\n');
13     }
14
15     std::cout << "Jour donné : " << jour << std::endl;
16     return 0;
17 }
```

Tant que l'entrée met `std::cin` dans un état invalide ou qu'elle n'est pas dans la plage de valeurs autorisées, on demande à l'utilisateur de saisir une nouvelle donnée. Vous reconnaitrez que ce code est tiré de notre fonction `entree_securisee`. Je l'ai extraite pour pouvoir ajouter des conditions supplémentaires. Nous verrons dans le prochain chapitre une façon d'améliorer ça.

Pour l'instant, retenez qu'il faut **vérifier qu'une donnée est cohérente** en plus de s'assurer qu'elle est du bon type.

14.2. À une condition... ou plusieurs

Écrire une fonction peut être plus ou moins simple, en fonction de ce qu'elle fait, mais beaucoup de développeurs oublient une étape essentielle consistant à **réfléchir au(x) contrat(s) que définit une fonction**. Un contrat est un accord entre deux parties, ici celui qui utilise la fonction et celui qui la code.

- La fonction mathématique `std::sqrt`, qui calcule la racine carrée d'un nombre, attend qu'on lui donne un **réel nul ou positif**. Si ce contrat est respecté, elle s'engage à retourner un **réel supérieur ou égal à zéro** et que **celui-ci au carré soit égal à la valeur donnée en argument**. Le contrat inhérent à la fonction `std::sqrt` se résume donc ainsi: « donne-moi un entier positif, et je te renverrai sa racine carrée ». Ça paraît évident, mais c'est important de le garder en tête.
- Quand on récupère un élément d'un tableau avec `[]`, il faut fournir un **index valide**, compris entre 0 et la taille du tableau moins un. Si on fait ça, on a la garantie d'obtenir un **élément valide**.
- Si l'on a une chaîne de caractères ayant **au moins un caractère**, on respecte le contrat de `pop_back` et on peut **retirer en toute sécurité le dernier caractère**.

On voit, dans chacun de ces exemples, que la fonction attend qu'on respecte une ou plusieurs conditions, que l'on nomme les **préconditions**. Si celles-ci sont respectées, la fonction s'engage en retour à respecter sa part du marché, qu'on appelle les **postconditions**. Les passionnés de mathématiques feront un parallèle avec les **domaines de définitions** des fonctions.

Le but de réfléchir à ces conditions est de produire du code plus robuste, plus fiable et plus facilement testable, pour toujours tendre vers une plus grande qualité de notre code. En effet, en prenant quelques minutes pour penser à ce qu'on veut que la fonction fasse ou ne fasse pas, on sait plus précisément quand **elle fonctionne** et quand **il y a un *bug***.

14.2.1. Contrats assurés par le compilateur

15. Le typage

Un exemple simple de contrat, que nous avons déjà vu sans même le savoir, s'exprime avec le typage. Si une fonction s'attend à recevoir un entier, on ne peut pas passer un `std::vector`. Le compilateur s'assure qu'on passe en arguments des types compatibles avec ce qui est attendu.

15.0.0.1. Avec `const`

Un autre type de contrat qu'on a déjà vu implique l'utilisation de `const`. Ainsi, une fonction qui attend une référence sur une chaîne de caractères constante garantie que celle-ci restera intacte, **sans aucune modification**, à la fin de la fonction. Cette postcondition est vérifiée par le compilateur, puisqu'il est impossible de modifier un objet déclaré comme étant `const`.

15.0.1. Vérifier nous-mêmes

Le compilateur fait une partie du travail certes, mais la plus grosse part nous revient. Pour vérifier nos contrats, nous avons dans notre besace plusieurs outils que nous allons voir.

15.1. Le développeur est un idiot

Vous souvenez vous du chapitre sur les tableaux ? Quand nous avons vu l'accès aux éléments avec les crochets, je vous ai dit de vérifier que l'indice que vous utilisez est bien valide, sous peine de comportements indéterminés. Si vous ne le faites pas, alors c'est de votre faute si le programme plante, car vous n'avez pas respectés une précondition. C'est ce qu'on appelle une **erreur de programmation**, ou ***bug***.

Dans le cas où ce genre de problème arrive, rien ne sert de continuer, mieux vaut arrêter le programme et corriger le code. Il existe justement un outil qui va nous y aider : **les assertions**.

Fichier à inclure

Pour utiliser les assertions, il faut inclure le fichier d'en-tête `<cassert>`.

Une assertion fonctionne très simplement. Il s'agit d'évaluer une condition quelconque. Si la condition est vraie, le programme continue normalement. Par contre, si elle se révèle être fausse, **le programme s'arrête brutalement**. Voyez par vous-mêmes ce qu'il se passe avec le code suivant.

```cpp
#include <cassert>

int main()
{
    // Va parfaitement fonctionner et passer à la suite.
    assert(1 == 1);
    // Va faire planter le programme.
    assert(1 == 2);
    return 0;
}
```

```
Assertion failed: 1 == 2, file d:\documents\visual studio
    2017\projects\zdscpp\zdscpp\main.cpp, line 8
```

Le programme plante et nous indique quel fichier, quelle ligne et quelle condition exactement lui ont posé problème. On peut même ajouter une chaîne de caractères pour rendre le message d'erreur plus clair. Cela est possible car, pour le compilateur, une chaîne de caractères est toujours évaluée comme étant `true`.

```cpp
#include <cassert>

int main()
{
    // Va parfaitement fonctionner et passer à la suite.
    assert(1 == 1 && "1 doit toujours être égal à 1.");
    // Va faire planter le programme.
    assert(1 == 2 && "Oulà, 1 n'est pas égal à 2.");
    return 0;
}
```

```
Assertion failed: 1 == 2 && "Oulà, 1 n'est pas égal à 2.", file
    d:\documents\visual studio
    2017\projects\zdscpp\zdscpp\main.cpp, line 8
```

Pour donner un exemple concret, on va utiliser `assert` pour vérifier les préconditions de l'accès à un élément d'un tableau. Ce faisant, nous éviterons les comportements indéterminés qui surviennent si nous violons le contrat de la fonction.

```cpp
#include <cassert>
#include <iostream>
```

```
3  #include <vector>
4
5  int main()
6  {
7      std::vector<int> const tableau { -4, 8, 452, -9 };
8      int const index { 2 };
9
10     assert(index >= 0 && "L'index ne doit pas être négatif.");
11     assert(index < std::size(tableau) &&
           "L'index ne doit pas être plus grand que la taille du tableau.");
12     std::cout << "Voici l'élément " << index << " : " <<
           tableau[index] << std::endl;
13
14     return 0;
15 }
```

Si, lors d'un moment d'inattention par exemple, nous avons utilisé un indice trop grand, alors le plantage provoqué par `assert` nous le rappellera et nous poussera ainsi à corriger le code. La documentation est, pour cela, une aide très précieuse, car elle indique comment réagit une fonction en cas d'arguments invalides. Dans la suite du cours, nous apprendrons à utiliser ces informations.

Mais je ne veux pas que mon programme plante. Pourquoi utiliser `assert`?

En effet, plus un programme est résistant aux erreurs et mieux c'est. Cependant, `assert` s'utilise non pas pour les erreurs de l'utilisateur mais bel et bien **pour celles du programmeur**. Quand le développeur est distrait et écrit du code qui résulte en comportements indéterminés (comme un dépassement d'indice pour un tableau), tout peut arriver et justement, **ce n'est pas ce qu'on attend du programme**. Cela n'a donc pas de sens de vouloir continuer l'exécution.

Par contre, les erreurs de l'utilisateur, par exemple rentrer une chaîne de caractères à la place d'un entier, ne sont absolument pas à traiter avec `assert`. On ne veut pas que notre programme plante brutalement parce que l'utilisateur s'est trompé d'un caractère.

15.1.1. Préconditions et assertions

Pour nos propres fonctions, le principe sera le même. Nous allons utiliser les assertions pour vérifier que les paramètres reçus respectent bien les contrats que nous avons préalablement définis.

Prenons un exemple bête avec une fonction chargée de renvoyer le résultat de la division de deux réels.

```
1  #include <cassert>
2  #include <iostream>
```

```
3
4  double division(double numerateur, double denominateur)
5  {
6      assert(denominateur != 0.0 &&
           "Le dénominateur ne peut pas valoir zéro.");
7      return numerateur / denominateur;
8  }
9
10 int main()
11 {
12     std::cout << "Numérateur : ";
13     double numerateur { 0.0 };
14     std::cin >> numerateur;
15
16     double denominateur { 0.0 };
17     // La responsabilité de vérifier que le contrat est respecté
           appartient au code appellant.
18     do
19     {
20         std::cout << "Dénominateur : ";
21         std::cin >> denominateur;
22
23     } while (denominateur == 0.0);
24
25     std::cout << "Résultat : " << division(numerateur,
           denominateur) << std::endl;
26     return 0;
27 }
```

La fonction `division` pose ici une précondition, qui est que le dénominateur soit non nul. Si elle est respectée, alors elle s'engage, en post-condition, à retourner le résultat de la division du numérateur par le dénominateur.

Au contraire, si le contrat est violé, le programme va s'interrompre sur le champ, car une division par zéro est mathématiquement impossible. Si le programme continuait, tout et n'importe quoi pourrait arriver.

15.2. Les tests unitaires à notre aide

15.2.1. Pourquoi tester ?

C'est une bonne question. Pourquoi ne pas se contenter de lancer le programme et de voir soi-même avec quelques valeurs si tout va bien ? Parce que **cette méthode est mauvaise**. Déjà, nous n'allons tester que quelques valeurs, en général des valeurs qui sont correctes, sans forcément penser aux valeurs problématiques. De plus, ces tests **sont consommateurs en temps**, puisqu'il faut les faire à la main.

À l'opposé, en écrivant des tests nous-mêmes, nous bénéficions de **plusieurs avantages**.

— Nos tests sont **reproductibles**, on peut les lancer à l'infini et vérifier autant que l'on veut. Ils sont déjà écrits, pas de perte de temps donc.
— Conséquence du point précédent, on peut **les lancer au fur et à mesure** que l'on développe pour **vérifier l'absence de régressions**, c'est-à-dire des problèmes qui apparaissent suite à des modifications, comme une fonction qui ne renvoie plus un résultat correct alors qu'auparavant, c'était le cas.
— Comme on ne perd pas de temps manuellement, on peut écrire **plus de tests**, qui permettent donc de détecter plus facilement et rapidement d'éventuels problèmes. Ceux-ci étant détectés plus tôt, ils sont **plus faciles à corriger** et le code voit sa qualité progresser.

15.2.2. Un test unitaire, qu'est-ce que c'est ?

Un test unitaire, abrégé TU est un morceau de code dont le seul but est de **tester un autre morceau de code**, afin de vérifier si celui-ci **respecte ses contrats**. Il est dit **unitaire** car il teste ce morceau de code **en toute indépendance** du reste du programme. Pour cela, les morceaux de code qui seront testés unitairement doivent être relativement courts et facilement isolable. Voici des exemples d'identifiants explicites qui vont vous aider à voir ce à quoi peut ressembler un test unitaire.

— `test_sinus_0_retourne_0`
— `test_cosinus_pi_retourne_moins1`
— `test_parenthesage_expression_simple_retourne_true`

Notez que les noms des fonctions sont **un peu long mais très clairs**. Cela est important, car quand certains tests échouent, s'ils ont un nom clair, qui définit sans ambiguïté ce que chaque test vérifie, il est plus facile de corriger ce qui ne va pas. Ainsi, si la fonction `test_cosinus_zero` échoue, vous savez où regarder. À l'inverse, un nom trop vague, trop générique, comme `test_cosinus`, ne nous donnerait aucune indication sur ce qui n'est pas correct.

Et si ce genre de test est nommé unitaire, c'est qu'il en existe d'autres types, que nous ne verrons pas dans le cadre de ce cours, mais dont vous entendrez parler au fur et à mesure que vous continuerez en programmation.

— Les **tests d'intégration**, qui vérifient que différents modules s'intègrent bien entre eux. On peut tester, par exemple, que le module de payement s'intègre bien au reste de l'application déjà développé.
— Les **tests fonctionnels**, qui vérifient que le produit correspond bien à ce qu'on a demandé. Des tests fonctionnels sur une calculatrice vérifieront que celle-ci offre bien les fonctionnalités demandées (addition, logarithme, etc) par rapport à ce qui avait été défini et prévu (si l'on avait demandé le calcul de cosinus, il faut que la calculatrice l'implémente).
— Les **tests d'UI** (interface utilisateur, de l'anglais *user interface*), qui vérifient, dans le cas d'une application graphique, que les boutons sont bien au bon endroit, les raccourcis, les menus et sous-menus, etc.
— Et encore de nombreux autres que je passerai sous silence.

15.2.3. Écrire des tests unitaires

Il existe de nombreuses bibliothèques pour écrire des tests unitaires en C++, certaines basiques, d'autres vraiment complètes, en bref, pour tous les goûts. Dans notre cas, nous n'avons pas besoin d'une solution aussi poussée. Nous allons écrire nos tests très simplement : quelques fonctions que nous lancerons dans le `main` pour vérifier que tout va bien.

Et pour illustrer cette section, nous allons tester un type que nous connaissons bien, `std::vector`. Bien entendu, il a déjà été testé bien plus complètement et profondément par les concepteurs de la bibliothèque standard, mais ce n'est pas grave.

Par exemple, pour tester que la fonction `push_back` fonctionne correctement, on va vérifier qu'elle augmente bien le nombre d'éléments du tableau de un, et que l'élément ajouté se trouve bien à la fin du tableau. On obtient un code qui ressemble à ça.

```cpp
#include <cassert>
#include <iostream>
#include <vector>

void test_push_back_taille_augmentee()
{
    std::vector<int> tableau { 1, 2, 3 };
    assert(std::size(tableau) == 3 &&
        "La taille doit être de 3 avant l'insertion.");

    tableau.push_back(-78);
    assert(std::size(tableau) == 4 &&
        "La taille doit être de 4 après l'insertion.");
}

void test_push_back_element_bonne_position()
{
    std::vector<int> tableau { 1, 2, 3, 4};
    int const element { 1 };

    tableau.push_back(element);

    assert(tableau[std::size(tableau) - 1] == element &&
        "Le dernier élément doit être 1.");
}

int main()
{
    test_push_back_taille_augmentee();
    test_push_back_element_bonne_position();

    return 0;
}
```

Si on lance ce code, on voit que rien ne s'affiche et c'est normal. Comme le `std::vector` fournit dans la bibliothèque standard est de qualité, on est assuré qu'il se comportera bien comme il doit. Cependant, si nous avions dû écrire nous-mêmes `std::vector`, alors ces tests nous auraient assurés de la qualité de notre code.

Vous pouvez vous entraîner en écrivant les tests unitaires des différents exemples et exercices des chapitres passés.

15.2.4. Les tests unitaires et les postconditions

Rappelez-vous, nous avons défini les postconditions comme étant les contrats que doit respecter une fonction si ses préconditions le sont. Nous pouvons, grâce aux tests unitaires, vérifier que la fonction fait bien son travail en lui fournissant des paramètres corrects et en vérifiant que les **résultats sont valides** et **correspondent à ce qui est attendu**.

15.3. [T.P] Gérer les erreurs d'entrée — Partie IV

Un autre genre de mauvaise pratique consiste à écrire des littéraux en dur dans le code. Un exemple, que j'ai moi-même introduit dans ce cours (honte à moi !), se trouve dans les précédents T.P sur la gestion des entrées.

```
1 std::cin.ignore(255, '\n');
```

On décide d'ignorer jusqu'à 255 caractères, mais que se passe-t-il si l'utilisateur décide d'écrire un roman en guise d'entrée ? Nous ne supprimons pas tous les caractères et notre fonction échoue à remplir sa mission : c'est **un *bug***, une **erreur de notre part**. Afin de le corriger, il faudrait vider le bon nombre de caractères. Comment faire ?

Dans le fichier d'en-tête `<limits>`, il existe un type `std::numeric_limits` qui permet, pour chaque type, comme `int`, `double` et autres, d'en obtenir quelques propriétés, telles la valeur maximale que peut stocker ce type, ou bien la valeur minimale. Comme ces valeurs peuvent varier en fonction de la plateforme et du compilateur, utiliser `std::numeric_limits` à la place de constantes écrites en dur est **un gage d'une meilleure robustesse et donc de qualité**.

```
1 std::numeric_limits</* type en question */>::/*fonction à appeller
      */
```

```
1 #include <iostream>
2 #include <limits>
3 
4 int main()
```

```
5  {
6      std::cout << "Valeur minimale d'un int : " <<
           std::numeric_limits<int>::min() << std::endl;
7      std::cout << "Valeur maximale d'un int : " <<
           std::numeric_limits<int>::max() << std::endl;
8
9      std::cout << "Valeur minimale d'un double : " <<
           std::numeric_limits<double>::min() << std::endl;
10     std::cout << "Valeur maximale d'un double : " <<
           std::numeric_limits<double>::max() << std::endl;
11
12     return 0;
13 }
```

Pour résoudre notre problème soulevé en introduction à cette section, il faut juste savoir **combien de caractères peut stocker au maximum `std::cin`**. Un `int`, par exemple, pourrait être trop petit et ne pas suffire, ou être trop grand et donc nous créer des problèmes. Un type spécifique, `std::streamsize`, est justement là pour ça.

En se basant sur ces informations, vous avez toutes les clés pour rendre notre fonction de gestion des entrées plus robuste.

Correction T.P Partie IV

15.4. L'exception à la règle

15.4.1. La problématique

Au fur et à mesure que vous coderez des programmes de plus en plus complexes, vous verrez qu'il existe plein de cas d'**erreurs extérieures**. Parfois, un fichier sera manquant ou supprimé pendant la lecture. D'autres fois, il n'y aura plus assez de mémoire pour stocker un gros tableau. Ce sont des situations qui n'arrivent que **rarement**, mais que le programme doit **être capable de gérer**.

Prenons un exemple très simple, avec l'ouverture et la lecture d'un fichier. Ce programme récupère un tableau de chaînes de caractères, lues depuis un fichier quelconque.

```
1  #include <fstream>
2  #include <iostream>
3  #include <string>
4  #include <vector>
5
6  std::vector<std::string> lire_fichier(std::string const &
       nom_fichier)
```

```
{
    std::vector<std::string> lignes {};
    std::string ligne { "" };
    std::ifstream fichier { nom_fichier };

    while (std::getline(fichier, ligne))
    {
        lignes.push_back(ligne);
    }

    return lignes;
}

int main()
{
    std::string nom_fichier { "" };
    std::cout << "Donnez un nom de fichier : ";
    std::cin >> nom_fichier;

    auto lignes = lire_fichier(nom_fichier);
    std::cout << "Voici le contenu du fichier :" << std::endl;
    for (auto const & ligne : lignes)
    {
        std::cout << ligne << std::endl;
    }

    return 0;
}
```

Si nous ne pouvons pas ouvrir le fichier parce que celui-ci n'existe pas, par exemple, alors il faut trouver un moyen de le **signaler à l'utilisateur**.

On peut modifier la fonction pour qu'elle renvoie un booléen, et le tableau sera un argument transmis par référence. C'est la façon de faire en C, mais elle comporte un gros défaut : on peut très bien **ignorer la valeur de retour** de la fonction et par là, ne pas détecter l'erreur potentielle. De plus, elle est peu élégante car elle surcharge le code de paramètres et de valeurs de retour inutiles - au sens où elles ne servent qu'à transmettre les erreurs et ne font pas partie du fonctionnement « théorique » du programme - ainsi que de code de vérification d'erreur lourd. Cette solution n'est donc pas bonne, puisqu'elle nuit à la robustesse et à l'expressivité du code.

On peut aussi afficher un message et demander de saisir un nouveau nom. C'est une solution parmi d'autres, mais elle implique de **rajouter des responsabilités** à notre fonction, puisque en plus de lire un fichier, elle doit maintenant gérer des entrées. De plus, si nous voulons changer nos entrées / sorties de la console vers un fichier, ou même un programme graphique, nous sommes **coincés**.

La solution consiste à utiliser un nouveau concept que je vais introduire : **les exceptions**.

15.4.2. C'est quoi une exception ?

C'est un mécanisme qui permet à un morceau de code, comme une fonction, de signaler au morceau de code qui l'a appelé que quelque chose d'exceptionnel - c'est-à-dire indépendant du développeur - s'est passé. On dit du premier morceau qu'il **lance / lève l'exception** (en anglais «*throw*») et que le second **la rattrape** (en anglais «*catch*»). Cela permet de gérer l'erreur dans le morceau de code appelant.

Contrairement aux assertions qui vont signaler la présence d'erreurs internes, de *bugs* à corriger, les exceptions s'utilisent pour les **erreurs externes**, erreurs qui peuvent arriver **même si votre programme est parfaitement codé**. Ne les voyez donc pas comme deux méthodes opposées, mais, bien au contraire, **complémentaires**.

Les exceptions sont notamment utilisées dans le cadre de la programmation par contrat : lorsqu'une fonction dont les préconditions sont vérifiées est dans l'incapacité de vérifier une de ses postconditions à cause d'un événement exceptionnel (un fichier inexistant par exemple), on lance une exception.

15.4.3. Lancement des exceptions dans 3, 2, 1...

Tout d'abord et comme pour beaucoup de choses en C++, il faut inclure un fichier d'en-tête, qui se nomme ici `<stdexcept>`. Ensuite, le reste est simple. On utilise le mot-clef `throw` suivi du type de l'exception qu'on veut lancer. Car, de même qu'il existe des types pour stocker des caractères, des entiers, des réels et autres, il existe **différents types d'exception** pour signaler une erreur de taille, une erreur de mémoire, une erreur d'argument, etc.

Celle que nous allons utiliser s'appelle `std::runtime_error` et représente n'importe quelle **erreur à l'exécution**. Elle attend une chaîne de caractères, le message d'erreur.

```cpp
#include <fstream>
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>

std::vector<std::string> lire_fichier(std::string const & nom_fichier)
{
    std::vector<std::string> lignes {};
    std::string ligne { "" };

    std::ifstream fichier { nom_fichier };
    if (!fichier)
    {
        // Si le fichier ne s'ouvre pas, alors on lance une exception pour le signaler.
        throw std::runtime_error("Fichier impossible à ouvrir.");
    }
```

```
18
19      while (std::getline(fichier, ligne))
20      {
21          lignes.push_back(ligne);
22      }
23
24      return lignes;
25  }
26
27  int main()
28  {
29      std::string nom_fichier { "" };
30      std::cout << "Donnez un nom de fichier : ";
31      std::cin >> nom_fichier;
32
33      auto lignes = lire_fichier(nom_fichier);
34      std::cout << "Voici le contenu du fichier :" << std::endl;
35      for (auto const & ligne : lignes)
36      {
37          std::cout << ligne << std::endl;
38      }
39
40      return 0;
41  }
```

> **i Tester l'ouverture d'un flux**
>
> Pour tester si notre fichier a bien été ouvert, il suffit d'un simple `if`.

Lancez le programme et regardez-le … planter lamentablement. Bouahaha, je suis cruel.

Que s'est-il passé? Lors du déroulement du programme, dès qu'on tombe sur une ligne `throw`, alors l'exécution de la fonction où l'on était **s'arrête immédiatement** (on dit que l'exception « suspend l'exécution ») et on revient à la fonction appelante. Si celle-ci n'est pas en mesure de gérer l'exception, alors on revient à la précédente encore. Quand enfin on arrive à la fonction `main` et que celle-ci ne sait pas comment gérer l'exception elle non plus, la norme C++ prévoit de tuer tout simplement le programme.

15.4.4. Attends que je t'attrape !

Il nous faut maintenant apprendre à rattraper (on dit aussi « catcher ») l'exception, afin de pouvoir la traiter. Pour cela, on utilise la syntaxe `try catch` (`try` étant l'anglais pour «essaye»).

Concrètement, si un code est situé dans un bloc `try` et que ce code lève une exception, **l'exécution du programme s'arrête à cette ligne et on part directement dans le bloc `catch`** pour traiter la-dite exception. Quant à ce dernier, il faut lui préciser, en plus du

mot-clef `catch`, le type d'exception à rattraper, par référence constante. Une fois une exception attrapée, on peut afficher le message qu'elle contient en appelant la fonction `what`.

```
#include <fstream>
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>

std::vector<std::string> lire_fichier(std::string const &
    nom_fichier)
{
    std::vector<std::string> lignes {};
    std::string ligne { "" };

    std::ifstream fichier { nom_fichier };
    if (!fichier)
    {
        throw std::runtime_error("Fichier impossible à ouvrir.");
    }

    while (std::getline(fichier, ligne))
    {
        lignes.push_back(ligne);
    }

    return lignes;
}

int main()
{
    std::string nom_fichier { "" };
    std::cout << "Donnez un nom de fichier : ";
    std::cin >> nom_fichier;

    try
    {
        // Dans le try, on est assuré que toute exception levée
        // pourra être traitée dans le bloc catch situé après.

        auto lignes = lire_fichier(nom_fichier);
        std::cout << "Voici le contenu du fichier :" << std::endl;
        for (auto const & ligne : lignes)
        {
            std::cout << ligne << std::endl;
        }
    }
    // Notez qu'une exception s'attrape par référence constante.
    catch (std::runtime_error const & exception)
```

```
46      {
47          // On affiche la cause de l'exception.
48          std::cout << "Erreur : " << exception.what() << std::endl;
49      }
50
51      return 0;
52  }
```

! L'exécution s'arrête lors d'une exception

Je l'ai écrit plus haut, mais je le remets bien lisiblement ici: dès qu'un appel à une fonction, dans le `try`, lève une exception, **le code suivant n'est pas exécuté et on file directement dans le `catch`**. C'est pour ça que le message `"Voici le contenu du fichier :"` n'est pas affiché.

Voilà, notre programme est maintenant capable de rattraper et de gérer cette situation exceptionnelle, bien qu'on se contente ici de simplement afficher la cause de l'erreur. On peut facilement imaginer un programme qui redemande de saisir un nom de fichier valide tant qu'on rencontre une erreur. Essayez donc de le faire.

Contenu masqué n°36

15.4.5. Attrapez-les tous !

Le `catch` qui suit le `try`, dans notre exemple, permet d'attraper des exceptions de type `std::runtime_error`. Mais certaines fonctions peuvent lancer plusieurs types d'exceptions. Il suffit cependant d'enchaîner les `catch` pour attraper autant d'exceptions différentes qu'on le souhaite.

Prenons l'exemple d'une fonction vue précédemment, permettant de convertir une chaîne de caractère en entier, `std::stoi`. Cette fonction peut lever deux types d'exception.

- Soit `std::invalid_argument` si la chaîne de caractère est invalide et ne peut pas être convertie.
- Soit `std::out_of_range` si le nombre à convertir est trop grand.

```
```

i Ordre des `catch`

L'ordre dans lequel vous mettez les `catch` n'a, ici, pas d'importance. En effet, si le `catch` est du même type que l'exception lancée, on rentre dedans, sinon on le saute.

Lancez le programme, modifiez la valeur donnée en argument à `std::stoi` et voyez par vous-mêmes comment on passe dans le `catch` attrapant l'exception lancée.

15.4.6. Un type pour les gouverner tous et dans le catch les lier

Il arrive, dans certains cas, qu'on ne veuille pas traiter différemment les exceptions qu'on attrape. Cela signifie t-il qu'on doit écrire autant de `catch` qu'il n'y a d'exceptions potentiellement lançables ? Non, on peut utiliser une **exception générique**.

Il existe un type, du nom de `std::exception`, qui décrit une exception générique, une erreur quelconque. Nous reviendrons plus tard sur les propriétés qui rendent ce résultat possible, mais, en attendant, retenez qu'on peut attraper n'importe quelle exception avec.

Son utilisation n'est absolument pas contraire à l'utilisation de type d'exception précis. On peut ainsi décider d'attraper certaines exceptions bien précises pour faire un traitement particulier, et attraper toutes les autres grâce à `std::exception` et suivre le même processus pour toutes celles-ci. Il n'y a qu'une chose à garder en tête : **le `catch` générique doit toujours se trouver en dernier**.

> **!** Les dangers du `catch` générique
>
> — Je répète, le `catch` générique doit **toujours être en dernier**. En effet, les exceptions sont filtrées dans l'ordre de déclaration des `catch`. Comme toutes les exceptions sont des versions plus spécifiques de `std::exception`, elles seront toutes attrapées par le `catch` générique et jamais par les éventuels autres `catch`.
>
> — Ne prenez pas la mauvaise habitude de toujours tout attraper. En effet, vous pourriez attraper par inadvertance des exceptions que vous ne gérez pas correctement. N'attrapez donc que les exceptions que vous souhaitez vraiment gérer. Lorsque vous êtes tentés de tout attraper, posez-vous bien cette question : *souhaité-je vraiment gérer toutes les erreurs ?*

15.5. [T.P] Gérer les erreurs d'entrée — Partie V

Puisque nous en sommes à améliorer notre T.P, que diriez-vous d'**afficher un meilleur message d'erreur** ? En effet, peu importe qu'on soit dans le cas d'une entrée de type invalide (une chaîne de caractères à la place d'un nombre) ou dans le cas d'une entrée incorrecte (8 alors qu'on demande un numéro entre 0 et 5), on affiche toujours le même message. Que diriez-vous de le rendre plus précis ? C'est une chose importante que de signaler clairement à l'utilisateur l'erreur qui s'est produite.

Pour nous aider, il existe deux fonctions. Celles-ci seront appliquées ici à `std::cin` mais sont disponibles également pour un flux de fichier ou de chaîne.

— `std::cin.eof()` renvoie `true` si le flux d'entrée est fermé, notamment dans le cas où on arrive à la fin d'un fichier. Cela peut aussi arriver quand vous tapez Ctrl+D dans le terminal sous GNU/Linux ou Ctrl+Z sous Windows.
— `std::cin.fail()` renvoie `true` si une erreur quelconque d'entrée apparaît. On va s'en servir notamment dans le cas où une entrée n'est pas du bon type.

> **Indice**
>
> N'oubliez pas que s'il y a fermeture de flux, il faudra lancer une exception puisque la lambda sera incapable de vérifier la postcondition « affecter une valeur correcte fournie par le flux à la variable ».

Avec ces éléments, vous êtes tous bons pour améliorer notre code. Bon courage.

Correction

On remarque qu'on a toujours le dilemme entre fonction ou gestion des erreurs plus personnalisée. Bientôt, nous trouverons une solution à ce problème.

> **Gestion d'erreur**
>
> Ceci n'est qu'un petit T.P illustratif. Notez bien que dans un programme réel, on ne laisserait pas l'exception faire planter le programme : on la gérerait de manière agréable pour l'utilisateur. Ici, typiquement, on pourrait afficher un message avant de fermer le programme, pour expliquer à l'utilisateur pourquoi on ne peut plus continuer.

15.5.1. En résumé

— Les entrées de l'utilisateur peuvent poser problème, c'est pour ça que nous les vérifions avec des conditions.
— Créer une fonction demande une réflexion sur ses préconditions et postconditions.
— Les assertions sont là pour les erreurs de programmation, les fautes du programmeur. On s'en sert pour les préconditions, notamment.
— Les tests unitaires présentent de nombreux avantages, notamment celui de vérifier les postconditions de nos fonctions, et donc leur bon fonctionnement.
— Les valeurs écrites en dur sont une mauvaise pratique et peuvent même créer des problèmes.
— Les exceptions nous permettent de signaler des erreurs externes, des situations sur lesquelles le programme n'a pas prise.
— Une exception se lève avec le mot-clé `throw`.
— Une exception se rattrape dans un bloc `catch`, qui suit immédiatement un bloc `try`.

- On peut avoir autant de bloc `catch` qu'on veut, un pour chaque type d'exception que l'on veut attraper.
- On peut attraper toutes les exceptions de manière générique en *catchant* `std::exception`.

Contenu masqué

Contenu masqué n°35 : Correction T.P Partie IV

```
```

Notez que le code est toujours dupliqué. Nous verrons progressivement comment améliorer ce problème dans les chapitres qui suivent. Retourner au texte.

Contenu masqué n°36

```
```

Retourner au texte.

Contenu masqué n°37 : Correction

```
```

Retourner au texte.

16. Des fonctions somme toute lambdas

Vous rappelez-vous du chapitre sur les algorithmes ? Nous avons découvert que nous pouvons personnaliser, dans une certaine mesure, leur comportement. Mais nous n'avions pas d'autres choix que d'utiliser les prédicats déjà fournis par la bibliothèque standard. Ce chapitre va corriger ça.

Nous allons découvrir comment écrire **nos propres prédicats** en utilisant **les lambdas**.

16.1. Une lambda, c'est quoi ?

Une lambda est une fonction, **potentiellement anonyme**, destinée à être utilisée pour des **opérations locales**. Détaillons ces différentes caractéristiques.

— Une lambda est **une fonction**, c'est-à-dire un regroupement de plusieurs instructions, comme nous l'avons vu au chapitre dédié. Rien de bien nouveau.
— Une lambda peut être **anonyme**. Contrairement aux fonctions classiques qui portent obligatoirement un nom, un identificateur, une lambda peut être utilisé **sans qu'on lui donne de nom**.
— Une lambda s'utilise pour des **opérations locales**. Cela signifie qu'une lambda peut être directement écrite dans une fonction ou un algorithme, au plus près de l'endroit où elle est utilisée, au contraire d'une fonction qui est à un endroit bien défini, à part.

> *i* **Les lambdas sont un concept universel**
>
> Si les syntaxes que nous allons découvrir sont celles de C++, les lambdas sont répandues et utilisées par de nombreux langages, comme C#, Haskell ou encore Python. Ce concept vient de la programmation fonctionnelle, une autre façon de programmer et dont C++ a repris plusieurs éléments.

16.1.1. Pourquoi a-t-on besoin des lambdas ?

En effet, c'est une question à se poser. Après tout, nous avons déjà les fonctions et donc rien ne nous empêche d'utiliser les fonctions pour personnaliser nos algorithmes. C'est tout à fait possible. Seulement voilà, les fonctions présentent deux inconvénients.

— D'abord, le compilateur n'arrive pas à optimiser aussi bien que si on utilise les lambdas. Ceci est l'argument mineur.

— L'argument majeur, c'est qu'on crée une fonction **visible et utilisable par tout le monde**, alors qu'elle n'a pour objectif que d'être **utilisée localement**. On dit que la fonction a une **portée globale**, alors qu'il vaudrait mieux qu'elle ait une **portée locale**.

Les lambdas sont donc une solution à ces deux inconvénients. Bien entendu, C++ vous laisse libre de vos choix. Si vous voulez utiliser des fonctions là où des lambdas sont plus adaptées, rien ne vous l'interdit. Mais s'il y a des outils plus efficaces, autant les utiliser.

16.2. Syntaxe

La syntaxe pour déclarer une lambda a des points communs avec celles des fonctions classiques, mais aussi quelques différences. Examinons la ensemble.

— La **zone de capture** : par défaut, une lambda est en totale isolation et ne peut manipuler aucune variable de l'extérieur. Grace à cette zone, la lambda va pouvoir modifier des variables extérieures. Nous allons y revenir.
— Les **paramètres de la lambda** : exactement comme pour les fonctions, les paramètres de la lambda peuvent être présents ou non, avec utilisation de références et/ou `const` possible.
— Le **type de retour** : encore un élément qui nous est familier. Il est écrit après la flèche `->`. Il peut être omis dans quelques cas, mais nous allons l'écrire à chaque fois dans une volonté d'écrire un code clair et explicite.
— Les **instructions** : je ne vous ferai pas l'affront de détailler.

16.2.1. Quelques exemples simples

Ainsi, l'exemple le plus simple et minimaliste que je puisse vous donner serait celui-ci.

Personnalisons la lambda pour qu'elle prenne un message à afficher.

Maintenant, que diriez-vous de la tester, avec un algorithme, pour en voir tout l'intérêt. Disons que l'on veut parcourir un tableau de chaînes de caractères.

J'ai formaté le code pour qu'il soit plus lisible, notamment en allant à la ligne pour les instructions de la lambda. Bien évidemment, vous n'êtes pas obligés de faire comme ça, mais sachez que c'est une écriture lisible et répandue.

L'exemple précédent montre une lambda qui ne prend qu'un paramètre. En effet, `std::for_each` attend un **prédicat unaire**, c'est-à-dire que l'algorithme travaille sur un seul élément à la fois. La lambda ne prend donc qu'un seul paramètre.

L'exemple suivant reprend le tri par ordre décroissant, qu'on a déjà vu avec des foncteurs, mais cette fois avec une lambda. Notez que celle-ci prend deux paramètres. C'est parce que `std::sort` attend un **prédicat binaire**, parce que l'algorithme manipule et compare deux éléments en même temps.

Continuons avec l'algorithme `std::find_if`. Celui-ci fonctionne exactement comme `std::find`, mais prend en plus un prédicat pour rechercher tout élément le vérifiant. Que diriez-vous, du coup, de chercher tous les éléments impairs d'une collection d'entiers ? Comme on regarde élément par élément, il s'agit d'un prédicat unaire.

Avez-vous noté, dans ces exemples, l'élégance des lambdas ? On l'a juste sous les yeux, on voit directement ce que fait l'algorithme. Et en plus, on ne pollue pas l'espace global avec des fonctions à usage unique.

16.3. Exercices

16.3.1. Vérifier si un nombre est négatif

Nous avons vu, dans le chapitre consacré aux algorithmes, `std::all_of`, qui permet de vérifier que tous les éléments respectent un prédicat. Il y a un algorithme semblable, `std::any_of`, qui s'utilise pareil mais regarde si au moins un élément vérifier le prédicat.

Le but est donc de dire si une collection d'entiers est entièrement positive ou non.

Correction vérifier si un nombre est négatif

16.3.2. Tri par valeur absolue

Rien de très compliqué, mais qui diriez-vous de trier un tableau dans l'ordre croissant en utilisant la valeur absolue des nombres ? Cela se fait en utilisant la fonction `std::abs`, disponible dans le fichier d'en-tête `<cmath>`.

Correction tri par valeur absolue

16.4. On va stocker ça où ?

Imaginons maintenant que l'on veuille appliquer une lambda plusieurs fois. Va-t-on copier-coller le code ? Vous connaissez déjà la réponse. Il suffit de **créer une variable**, tout simplement, comme nous l'avons fait jusque-là. La façon la plus simple est d'utiliser `auto`.

> **Syntaxe et question de goût**
>
> On peut, comme le démontre le code, utiliser la syntaxe de C++ moderne, à base d'accolades `{}`, ou bien utiliser l'initialisation héritée du C à base de égal `=`. Tout est une question de goût.
> J'ai choisi la deuxième syntaxe car je la trouve plus lisible, mais vous restez libres de vos choix.

Stocker une lambda permet de garder les avantages de localité tout en évitant de répéter du code, dans le cas où elle est utilisée plusieurs fois.

> Par curiosité, quel est le type explicite d'une lambda ?

C'est un type complexe qui n'est pas spécifié par la norme. Le compilateur transforme la lambda en autre chose, et cet « autre chose » dépend donc du compilateur que vous utilisez. Les lambdas nous donnent donc un excellent exemple de l'intérêt et de la simplicité apportée par `auto`.

16.5. Paramètres génériques

Tout comme il est possible de laisser le compilateur deviner le type de retour, on peut lui laisser le soin de **déduire les paramètres de la lambda**, en utilisant `auto` à la place du type explicite des paramètres. Bien entendu, le fait de l'utiliser pour certains paramètres **ne nous empêche absolument pas d'avoir des paramètres au type explicitement écrit**.

Du coup, grâce à ce nouveau pouvoir que nous avons, nous sommes en mesure d'utiliser un même lambda pour afficher deux collections de types différents.

16.6. [T.P] Gérer les erreurs d'entrée — Partie VI

Avec les connaissances que vous avez, nous pouvons reprendre le code où nous l'avons laissé et commencer à réduire la duplication en utilisant une lambda. Faites-moi une entrée sécurisée mais sans dupliquer le code. Bonne chance.

```
#include <iostream>

int main()
{
    int jour { 0 };
    std::cout << "Quel jour es-tu né ? ";
    // Entrée.

    int mois { 0 };
    std::cout << "Quel mois ? ";
    // Entrée.

    int annee { 0 };
    std::cout << "Quelle année ? ";
    // Entrée.

    double taille { 0.0 };
    std::cout << "Quelle taille ? ";
    // Entree.

    std::cout << "Tu es né le " << jour << "/" << mois << "/" <<
        annee << " et tu mesures " << taille << "m." << std::endl;
    return 0;
}
```

Correction T.P partie VI

Le code est déjà mieux qu'avant. Il reste cependant un problème : la lambda est locale à la fonction, donc si nous voulons utiliser notre fonction sécurisée à plein d'endroits différents, il va falloir trouver une autre solution. Nous verrons cela très bientôt.

16.7. [T.P] Gérer les erreurs d'entrée — Partie VII

Eh oui, nous allons encore enchaîner deux T.P d'affilée ! Je l'ai découpé pour que vous puissiez l'aborder avec plus de facilité.

Relisons le chapitre sur les erreurs. Nous y avions vu une méthode pour non seulement vérifier que l'entrée ne faisait pas planter `std::cin`, mais en plus que la valeur donnée avait un sens.

Nous pouvions ainsi vérifier qu'un jour était compris entre 1 et 31, ou qu'un mois est inclus entre 1 et 12. Mais pour cela, nous avions dû extraire notre code de la fonction `entree_securisee`.

Cependant, nous venons de voir que les lambdas acceptent des paramètres génériques avec `auto`. Alors, soyons fous, pourquoi ne pas **essayer de passer une autre lambda** ?

Ça marche parfaitement ! Et si nous utilisions ce concept puissant en combinaison avec notre T.P précédent ? On pourrait ainsi vérifier que le mois et le jour sont corrects, ou bien même que la personne est née après 1900. Vous avez à présent tous les indices qu'il vous faut pour résoudre ce cas. Bonne chance.

Correction T.P partie VII

16.8. Capture en cours...

16.8.1. Capture par valeur

Abordons maintenant cette fameuse zone de capture. Par défaut, une lambda est en totale isolation du reste du code. Elle ne peut pas, par exemple, accéder aux variables de la fonction dans laquelle elle est écrite... sauf si on utilise la zone de capture pour dire **à quels objets la lambda peut accéder**. On peut bien sûr en capturer plusieurs, sans problème.

Remarquez que si l'on supprime `un_booleen` de la zone de capture, le code ne compile plus. On parle donc de zone de capture car **la lambda a capturé notre variable**, afin de l'utiliser. Cette capture est dite **par valeur**, ou **par copie**, car chaque variable est copiée.

! Pour être exact

Il est possible, notamment si vous utilisez GCC ou MinGW (avec Qt Creator, entre autre) que vous avez deux warnings comme ceux ci-dessous.

Selon la norme, il y a effectivement certains cas où l'on n'est pas obligé de capturer une variable pour l'utiliser. Cependant, ces cas ne sont pas très nombreux, difficiles à expliquer à ce stade de votre apprentissage et présents surtout par volonté de compatibilité avec l'ancienne norme C++03.
Visual Studio 2017 ne respecte pas la norme sur ce point et donc nous sommes **obligés de capturer toutes les variables que nous voulons utiliser**, si nous voulons que notre code compile avec. Voilà pourquoi je vous invite, très exceptionnellement, à **ignorer**

le warning et à utiliser les mêmes codes que dans ce cours.

16.8.1.1. Modifier une variable capturée

Reprenons le code précédent et modifions-le pour ne garder qu'un entier modifiable (sans `const` donc). Capturez-le et essayer de le modifier. Vous allez obtenir une belle erreur, et je l'ai fait exprès.

Je ne comprends pas pourquoi je n'ai pas le droit de les modifier. Si c'est une capture par valeur, alors je manipule des copies bien distinctes des objets originaux, il ne devrait pas y avoir de problème. Pourquoi ?

Il y a plusieurs réponses à cette question. L'une est apportée par Herb Sutter, un grand programmeur, très investi dans C++ et l'écriture de sa norme.

> Le fait qu'une variable capturée par copie ne soit pas modifiable semble avoir été ajouté parce qu'un utilisateur peut ne pas se rendre compte qu'il modifie une copie et non l'original.
>
> *Herb Sutter*

Une autre sera détaillées plus tard dans le cours, car il nous manque encore des notions pour aborder ce qui se passe sous le capot.

Retenez qu'on ne peut pas, par défaut, modifier une variable capturée par copie. Heureusement, il suffit d'un simple mot-clef, `mutable`, placé juste après la liste des arguments et avant le type de retour, pour rendre cela possible.

Comme vous le montre le résultat de ce code, **la modification est locale à la lambda**, exactement comme avec les fonctions. Mais ce n'est pas le seul mode de capture existant, comme vous vous en doutez peut-être.

16.8.2. Capture par référence

Le principe est exactement le même, sauf que cette fois, on a affaire directement à la variable capturée et non à sa copie et pour ce faire, on rajoute l'esperluette `&` devant. Cela permet, comme pour les paramètres, d'éviter une copie potentiellement lourde, ou bien de sauvegarder les modifications faites à la variable.

Notez qu'on peut capturer certaines variables par référence, d'autres par valeur. C'est exactement comme pour les paramètres d'une fonction, donc cela ne doit pas vous surprendre.

- Les lambdas sont des fonctions anonymes, destinées à être utilisées localement.
- Elles réduisent la taille du code, le rendent plus lisible et évitent de créer des fonctions à portée globale alors qu'elles sont à usage local.
- Les paramètres peuvent être explicitement typés ou automatiquement déduit en utilisant `auto`.
- La véritable nouveauté des lambdas, par rapport aux fonctions, est la zone de capture, qui permet de manipuler, par référence et/ou valeur des objets de la portée englobante.
- On peut capturer explicitement ou bien certaines variables, ou bien capturer toutes celles de la portée englobante.

Contenu masqué

Contenu masqué n°38 : Correction vérifier si un nombre est négatif

Retourner au texte.

Contenu masqué n°39 : Correction tri par valeur absolue

Ne soyez par surpris par la sortie du programme. Rappelez-vous que nous avons trié le résultat en utilisant la valeur absolue, mais que l'affichage nous montre les éléments triés, mais originaux. Retourner au texte.

Contenu masqué n°40 : Correction T.P partie VI

Retourner au texte.

Contenu masqué n°41 :
Correction T.P partie VII

Même si le problème soulevé dans le T.P précédent, à savoir qu'on ne peut pas utiliser la lambda ailleurs, reste vrai, notez comme nous avons amélioré notre fonction pour lui faire vérifier que non seulement la valeur est du bon type, mais qu'elle est correcte. Retourner au texte.

17. Envoyez le générique !

Nous avons déjà vu avec les lambdas qu'il est possible d'écrire du code **générique**. On en a vu l'intérêt, notamment pour éviter la duplication de code. Par exemple, notre fonction `entree_securisee` a pu être écrite une seule fois pour tous les types au lieu de notre ancienne version, qu'il fallait écrire une fois pour les `int` et une fois pour les `double`.

On a donc un double intérêt : le code est **plus facilement maintenable** puisque les modifications se font une seule fois et ainsi on évite de réécrire une nouvelle version pour chaque nouveau type.

Le problème, c'est qu'on ne sait faire ça qu'avec des lambdas. Ça serait dommage de se limiter à elles seules. Heureusement, en C++, il est possible de rendre n'importe quelle fonction générique. C'est justement ce que nous allons voir.

17.1. Quel beau modèle !

Pour créer des fonctions génériques en C++, on écrit ce que l'on appelle des **templates** (en français « modèle » ou « patron ») : au lieu d'écrire directement une fonction, avec des types explicites, on va en écrire une version générique avec des **types génériques**. C'est ensuite le compilateur qui, quand on en aura besoin, utilisera ce modèle pour créer une fonction avec les bons types. On dit qu'**il instancie le modèle**.

Allons droit au but, voici la syntaxe générale, qui ressemble beaucoup à celle utilisée pour créer une fonction.

```
```

Notez aussi l'apparition de deux mot-clés.

— Le mot-clé `template` prévient le compilateur qu'on n'écrit pas une fonction classique mais un modèle.
— Le mot-clé `typename` permet de déclarer un nouveau **nom de type générique**.

Le principe est le suivant : au lieu d'utiliser uniquement des types précis (`int`, `double`, `std::string`, etc) comme on en a l'habitude, on va définir des noms de type quelconques, c'est-à-dire **pouvant être remplacés par n'importe quel type** au moment de l'utilisation de la fonction. Le compilateur va ensuite créer la fonction à notre place, en remplaçant les types génériques par les types réels des arguments qu'on va lui passer.

Parallèle

On peut ici remarquer un parallèle avec les concepts de paramètre et d'argument.
— Le type générique **représente** ce que le template attend comme type, jouant un rôle similaire à celui d'un paramètre de fonction.
— Le type réel est ce qui va être utilisé pour créer la fonction en remplaçant le type générique, à l'instar de l'argument qui est la valeur avec laquelle est appelée la fonction.

Un exemple valant mieux que mille mots, voici un template permettant d'afficher tous les éléments d'un `std::vector` dans la console. On a déjà vu une telle fonction, mais ici la différence est qu'elle va fonctionner **quel que soit le type des éléments dans le tableau**.

Quand le compilateur voit ce template, il comprend qu'il faut l'instancier trois fois, car il y a trois appels avec trois types différents. Bien sûr, la mécanique interne dépend du compilateur, mais on peut imaginer qu'il génère un code comme celui-ci.

Principe de la programmation générique

On peut remarquer ici un principe de base de ce qu'on appelle la **programmation générique** : on se fiche du type réel des éléments du tableau, du moment qu'ils sont affichables !
En programmation générique, ce qui compte, ce n'est pas le type exact, mais c'est **ce que l'on peut faire avec**.

On peut faire encore mieux : on se fiche du type réel du conteneur, **du moment qu'il est itérable**. Essayons d'appliquer ce principe et d'écrire une fonction `afficher` qui marche avec n'importe quel type de conteneur itérable.

Pour vous aider à bien visualiser ce qui se passe, voici ce que peut produire le compilateur quand il a fini d'instancier le modèle.

Magique, non ?

17.1.1. La bibliothèque standard : une fourmilière de modèles

En fait, on a vu et utilisé un **nombre incalculable de fonctions génériques dans la bibliothèque standard** : celles renvoyant des itérateurs (`std::begin`, `std::end`, etc), les conteneurs, les algorithmes, etc. En effet, toutes ces fonctions peuvent s'utiliser avec des arguments de types différents. Pour `std::begin`, par exemple, il suffit que le type passé en argument ait la capacité de fournir un itérateur sur le premier élément.

Le principe même des templates est d'oublier le type pour se concentrer sur **ce que peut faire ce type**, et c'est très intéressant pour la bibliothèque standard puisqu'elle a vocation à être utilisée dans des contextes variés. En fait, sans les templates, il serait très difficile de concevoir la bibliothèque standard du C++. Cela nous donne un exemple concret de leur intérêt.

17.1.2. Un point de vocabulaire

Lorsqu'on parle de templates, on utilise un vocabulaire spécifique pour pouvoir parler de chaque étape dans leur processus de fonctionnement. Reprenons-le pour bien que vous le mémorisiez.

- Le template est la description générique de ce que fait la fonction.
- Lorsque le compilateur crée une fonction à partir du modèle pour l'utiliser avec des types donnés, on parle **d'instanciation**.
- Lorsqu'on appelle la fonction instanciée, le nom ne change pas de ce dont on a l'habitude. On parle toujours **d'appel de fonction**.

17.2. [T.P] Gérer les erreurs d'entrée - Partie VIII

Nous avons déjà, dans la partie précédente, éliminé la duplication de code en utilisant une lambda. Nous avons également appris à transmettre une fonction, ce qui permet d'ajouter des vérifications supplémentaires. Cependant, notre code était dans une portée locale. Maintenant, vous disposez de toutes les clés pour corriger ce problème.

Le code à sécuriser est toujours le même.

```
#include <iostream>

int main()
{
    int jour { 0 };
    std::cout << "Quel jour es-tu né ? ";
    // Entrée.

    int mois { 0 };
    std::cout << "Quel mois ? ";
    // Entrée.

    int annee { 0 };
```

```
14      std::cout << "Quelle année ? ";
15      // Entrée.
16
17      double taille { 0.0 };
18      std::cout << "Quelle taille ? ";
19      // Entree.
20
21      std::cout << "Tu es né le " << jour << "/" << mois << "/" <<
            annee << " et tu mesures " << taille << "m." << std::endl;
22      return 0;
23  }
```

Correction T.P partie VIII

Et voilà! Maintenant, `entree_securisee` est générique et utilisable à une portée globale, on a tout ce que l'on voulait! Comme vous voyez, on est de plus en plus efficace pour éviter la duplication de code.

17.3. Instanciation explicite

Il arrive parfois que le compilateur ne puisse pas déterminer avec quels types il doit instancier un template, typiquement lorsque les types génériques n'apparaissent pas dans les paramètres de la fonction. C'est en particulier le cas lorsqu'il n'y a tout simplement pas de paramètre.

Par exemple, imaginons que notre fonction `entree_securisee`, au lieu de prendre la variable à modifier en paramètre, crée une variable du bon type et renvoie le résultat.

Jusque-là, pas de problème. Essayons d'appeler la fonction.

Et là, c'est le drame : impossible de compiler, il y a des erreurs remontées. Visual Studio vous sortira une erreur comme celle ci-dessous (en français ou en anglais, tout dépend de votre langue d'installation).

Si vous utilisez MinGW ou GCC, vous aurez des erreurs comme celles-ci.

Leur origine est simple : le compilateur n'a **aucun moyen de savoir avec quel type instancier le template**. Ce cas se présente aussi quand, pour un type générique attendu, on en donne plusieurs différents.

Dans le deuxième cas, il y a ambiguïté. Pour la lever, il suffit de préciser explicitement les types devant être utilisés pour l'instanciation, en les mettant entre chevrons juste avant les parenthèses de paramètres. La syntaxe ne vous est pas totalement inconnue, puisqu'elle est proche de celle utilisée pour les conteneurs (`std::vector<int>`, `std::list<double>`, etc).

> **Instanciation explicite de template**
>
> Cette précision explicite des types pour l'instanciation sert à lever les ambiguïtés, mais elle peut aussi s'utiliser dans d'autres cas. Par exemple, vous pouvez tout à fait préciser les types si vous trouvez ça plus lisible, même s'il n'y a pas d'ambiguïté.

17.4. Ce type là, c'est un cas !

On a vu que les templates sont une fonctionnalité très puissante pour écrire du code **générique**, c'est-à-dire ne dépendant pas des types des paramètres mais des **services**, des **fonctionnalités** proposées par ces types. Cependant, parfois, on aimerait que, pour un type particulier, la fonction se comporte différemment.

Revenons au template `afficher` qu'on a défini précédemment.

Ce code donne la sortie suivante.

Le résultat est assez désagréable à lire lorsqu'on utilise la fonction avec le type `std::string`. Heureusement, on a un moyen très simple de modifier le comportement de notre template pour un type de paramètre en particulier. C'est une notion que nous avons abordée dans le chapitre sur les fonctions : **la surcharge**. On va surcharger la fonction `afficher` pour les chaînes de caractères.

Ce qui donne un résultat bien plus lisible en sortie.

On a donc un template qui permet d'afficher toutes sortes de collections et un traitement à part pour les `std::string`. Bien entendu, on peut créer d'autres surcharges pour d'autres types, si on veut qu'ils aient un comportement bien précis.

Surcharge prioritaire

Le changement du comportement par surcharge fonctionne car le compilateur prend **en priorité les fonctions non-templates** lors du choix de la bonne surcharge. En effet, a priori, il devrait y avoir ambiguïté (et donc erreur), car il y a deux fonctions possibles : celle générée à partir du modèle et celle écrite explicitement.
Heureusement, le choix a été fait par les concepteurs du C++ d'accorder la priorité aux fonctions écrites explicitement. Cela permet donc de rendre valide ce genre de code.

17.4.1. En résumé

- Les templates permettent de gagner en généricité et donc de réduire la duplication de code.
- C'est une généralisation du principe de généricité, qu'on a déjà observé avec les lambdas.
- Cela permet une plus grande flexibilité ainsi que l'écriture de fonctions génériques dans la portée globale.
- La bibliothèque standard regorge de templates, car la généricité en est un aspect important.
- On peut préciser, ou non, explicitement les types lors de l'instanciation, mais certains cas rendent cette précision obligatoire.
- On peut surcharger une fonction générique pour personnaliser son comportement pour certains types explicitement donnés.

Contenu masqué

Contenu masqué n°42 : Correction T.P partie VIII

Templates et algorithmes standards

Avez-vous noté les deux lignes du début de notre modèle ?

Grâce aux templates, nous pouvons passer une fonction sans effort ni difficulté. C'est de la même manière que procède la bibliothèque standard pour les algorithmes personnalisables.

Enfin, grâce à la surcharge, on peut combiner les deux versions de notre fonction, pour choisir, au choix, de passer ou non un prédicat.

Retourner au texte.

18. De nouvelles structures de données

Lors de ce cours, nous avons eu l'occasion d'utiliser des données sous de nombreuses formes : types de base, chaînes de caractères, collections de données, etc. Cependant, toutes ces formes de données sont finalement assez rudimentaires, et on souhaite parfois avoir plus de souplesse dans la manière dont sont structurées nos données.

Dans ce chapitre, nous allons justement découvrir de **nouvelles structures de données** qui vont nous offrir plus de flexibilité dans nos programmes. Pour chaque structure, nous verrons comment elle s'utilise et en quoi elle est utile.

18.1. struct — Un agrégat de données

En C++, il existe une structure de données permettant de regrouper plusieurs données en un seul bloc. Dans leur grande originalité, elle s'appelle… **structure**.

Point de vocabulaire

À partir de maintenant, j'emploierai le terme « structure » pour désigner l'agrégat que nous allons introduire et l'expression « structure de données » pour désigner les formes complexes de données de manière générale.

Les structures sont très utiles pour regrouper des données qui ont un lien entre elles et travailler dessus de manière uniforme, et non sur chaque donnée séparément. Voyons un exemple concret avec un petit exercice.

18.1.1. Stockage d'informations personnelles[1]

Essayez de créer un programme qui demande des informations sur une personne et les stocke dans un fichier `Prenom.Nom.csv` sous le format suivant.

```

```

En sortie, la console devrait ressembler à quelque chose comme ci-dessous.

```

```

1. En total accord avec le RGPD.

Le programme va ensuite enregistrer ces informations dans un fichier `Clem.Lagrume.csv`, organisé comme suit.

Bonne chance.

Correction

18.1.2. Analyse de l'exercice

On a réussi à mener à bien l'exercice, mais on peut remarquer des pistes d'amélioration :

- Les fonctions `demander_infos` et `enregistrer_infos` prennent beaucoup de paramètres. Mais, à la limite, c'est du détail.
- Plus grave, si on ajoute une information sur une personne, il faudra modifier les prototypes de toutes nos fonctions ! Cela porte gravement atteinte à l'évolutivité de nos programmes.
- Les données que l'on manipule sont liées et pourraient s'utiliser en tant qu'ensemble.

18.1.3. Introduction aux structures

La solution est justement d'utiliser une structure. Cela va nous permettre de créer un type, que l'on va par exemple appeler `InformationsPersonnelles`, regroupant l'intégralité des informations que l'on souhaite enregistrer. La syntaxe de création est la suivante.

- L'identifiant de la structure suit **les mêmes règles de nommage** que les identifiants de variables et de fonctions.
- En termes de convention, nous suivrons une convention appelée CamelCase : on colle les mots en commençant chaque mot par une majuscule. Il en existe d'autres, mais nous utiliserons celle-là car elle permet facilement de distinguer les variables et les types.
- La définition d'une structure doit **se terminer par un point virgule**. Il est fréquent de l'oublier.

Par exemple, une structure représentant les données personnelles d'une personne, dont on pourrait se servir dans l'exercice précédent, peut être définie comme suit.

18.1.3.1. Initialisation

Pour initialiser une structure, nous disposons de plusieurs façons de faire, qui ne vont pas vous surprendre, puisque nous les connaissons déjà pour les variables « classiques ».

Notez que, lorsqu'on initialise avec des valeurs choisies, il faut les donner dans l'ordre dans lequel elles sont définies dans la structure. On ne peut pas donner un entier comme première valeur, si celle-ci est déclarée dans la définition comme étant une chaîne de caractères.

18.1.3.2. Voici les membres

Pour manipuler un membre, c'est-à-dire une variable appartenant à la structure, il suffit d'utiliser la syntaxe `structure.membre`, qui fonctionne tant en lecture qu'en écrire.

Bien entendu, comme toujours, la modification d'un membre n'est possible que si la variable n'est pas déclarée comme étant `const`.

18.1.4. Exercice

Sans plus attendre, modifions le code de l'exercice précédent pour utiliser une structure. C'est simple, modifiez-le pour utiliser une structure au lieu de se trimballer plein de variables.

Correction

Et voilà ! Le code est **plus lisible** et **plus évolutif**. Vous pouvez essayer d'ajouter une information à enregistrer, par exemple la couleur préférée ou la date de naissance, et vous verrez que vous aurez **uniquement des ajouts à faire**. C'est un critère important d'évolutivité du code. En particulier, les prototypes des fonctions **ne changeront pas**, seules les implémentations changeront. Du point de vue de l'utilisation, c'est transparent, on ne voit rien de changé, tout le code les utilisant n'aura donc pas à être réécrit.

> **Raccourci**
>
> Remarquez que, maintenant qu'on peut directement renvoyer le résultat de `demander_infos`, on n'est plus obligé de prendre en paramètres des références vers les variables à modifier. Cela permet donc d'écrire, non sans élégance :

18.2. std : :tuple — Une collection hétérogène

Les structures ne sont pas le seul moyen de représenter un ensemble de données de types variés. Il existe un autre outil nommé `std::tuple`, qui est une **collection hétérogène de taille fixée**. Hétérogène, car on peut stocker des informations de types différents (comme `struct`), mais dont la taille ne varie pas (comme `std::array`).

Fichier à inclure

Pour utiliser les tuples, il faut inclure le fichier `<tuple>`.

18.2.1. Déclaration

Pour préciser les types des différents éléments, on utilise une syntaxe bien connue. Par exemple, si on veut créer un tuple pour contenir des informations personnelles, comme dans l'exercice précédent, c'est très simple.

N'oubliez pas le `s`

Je vous rappelle qu'avec les littéraux, pour que le type déduit soit bien `std::string`, vous devez rajouter le `s` après chaque littéral (ainsi que la ligne `using namespace std::literals;`). C'est ici obligatoire pour que l'on crée un `std::tuple` contenant des `std::string` et non des chaînes héritées du C.

On a aussi une fonction `std::make_tuple` permettant de créer un tuple en déduisant les types.

18.2.2. Accès aux éléments

Comme pour toute collection, il faut aussi être capable d'accéder aux éléments. Pour cela, on utilise la fonction `std::get`. Celle-ci prend entre chevrons l'indice de l'élément auquel on veut accéder.

Dans le cas où tous les éléments d'un `std::tuple` ont un type unique, et **seulement dans ce cas**, on peut aussi y accéder en donnant à `std::get` le type de l'élément désiré.

Si vous essayez d'utiliser cette syntaxe alors qu'il y a plusieurs éléments de même type, le code ne compilera tout simplement pas.

18.2.3. `std::tuple` et fonctions

On peut tout à fait créer une fonction qui renvoie un `std::tuple`. C'est même une astuce qu'on utilise pour contourner la limite qu'impose C++ de ne pouvoir retourner qu'une seule valeur à la fois. Il suffit de déclarer que la fonction renvoie un `std::tuple` et de préciser, entre chevrons, les types de ses éléments.

Notez, dans l'exemple suivant, la syntaxe alternative et concise pour renvoyer un `std::tuple`, introduite depuis C++17.

18.2.4. Équations horaires

Vous voulez un exemple concret d'utilisation des `std::tuple` ? Vous aimez la physique ? Vous allez être servi. Nous allons coder une fonction retournant les coordonnées $(x; y)$ d'un objet lancé à une vitesse v_0 et à un angle α, en fonction du temps t.

> **Point de vocabulaire**
>
> C'est ce qu'on appelle **l'équation horaire** de l'objet. Cette notion est abordée, en France, au lycée en Terminale. Ne vous inquiétez pas si vous ne connaissez pas, nous n'allons pas du tout rentrer dans les détails.

Un cours de physique vous expliquera qu'on obtient la position de l'objet en fonction du temps t grâce à deux équations.

$$\begin{cases} x(t) = v_0 \, . \, cos \, \alpha \, . \, t \\ y(t) = -g \, . \, \frac{t^2}{2} + v_0 \, . \, sin \, \alpha \, . \, t \end{cases}$$

Donc, si on traduit tout ça en C++, on obtient quelque chose comme ça.

> Pourquoi s'encombrer des tuples ? Que dois-je choisir ?

Il est vrai que `std::tuple` semble présenter plus d'inconvénients qu'autres choses. Le code n'est pas très lisible avec tous ces `std::get`. De plus, contrairement à une structure dont l'accès à un élément est clair (comme `informations.prenom`), il ne saute pas aux yeux que `std::get<1>(informations)` représente potentiellement un prénom.

En fait, chaque solution a ses avantages et ses inconvénients, et on va donc choisir en conséquence selon les situations.

- Le tuple permet de rapidement avoir une collection hétérogène **sans avoir à définir un nouveau type**. Il est particulièrement utile lorsque l'on a besoin d'une collection hétérogène **uniquement pour une portée locale**, dans le corps d'une fonction par exemple. Par contre, la syntaxe d'utilisation, notamment pour l'accès aux éléments, est peu lisible.
- La structure permet une **plus grande expressivité**, notamment par le fait que les éléments ont des noms. Elle est particulièrement adaptée pour une utilisation sur une portée plus globale.

En fait, le tuple est à la structure ce que la lambda est à la fonction, finalement. C++ vous laisse libre, c'est vous qui choisirez en fonction des cas.

18.3. std : :unordered_map — Une table associative

Depuis le début de ce cours, nous avons vu les conteneurs `std::vector`, `std::array` et `std::string`. Nous avons également, très brièvement, survolé `std::list`. Maintenant, j'aimerais vous en présenter d'autres, qui sont un peu différents, mais très utiles.

Le premier conteneur se nomme `std::unordered_map`. Ce nom peut être traduit par « tableau associatif », « table associative » ou « carte associative », car son principe est **d'associer une valeur à une clé**. Ce principe, nous le connaissons déjà avec `std::vector`, qui associe une valeur quelconque à une clé entière. Ainsi, quand vous faites `tableau[4]`, vous utilisez une clé de type entier et qui vaut `4`. Grâce à `std::unordered_map`, nous avons le choix du type de la clé.

Fichier à inclure

Pour utiliser ce nouveau conteneur, pensez à inclure le fichier `<unordered_map>`.

18.3.1. Un dictionnaire de langue Zestienne

Imaginez que vous vouliez créer un dictionnaire. Comme pour la version papier, nous voulons associer un mot à un ensemble de phrases. En C++, on peut traduire ça par une `std::unordered_map` dont la clé est de type `std::string`, ainsi que la valeur. On précise le type de la clé et de la valeur entre chevrons, car notre conteneur utilise les templates.

Pour afficher le contenu de notre conteneur, on va récupérer les éléments un par un. On peut faire ça avec une simple boucle `for`, comme on en a l'habitude. Chaque élément est un `std::pair<T, U>`, où `T` est de même type que la clé et `U` de même type que la valeur. Ce type n'est rien d'autre qu'un cas particulier de tuple à deux valeurs uniquement.

Ce type fournit deux propriétés qui nous intéressent, qui sont `first` et `second`, renvoyant respectivement la clé et la valeur récupérée dans la `std::unordered_map`. On a donc maintenant toutes les informations pour écrire la boucle et afficher les éléments de notre tableau associatif.

Petite précision

Je vous ai expliqué ce qu'est `std::pair`, mais j'ai utilisé `auto` à la place. Le code est ainsi plus court et plus lisible. Vous êtes néanmoins libre d'écrire explicitement `std::pair<std::string, std::string>` si vous le souhaitez.

18.3.2. Toujours plus de clés

Bien entendu, si notre conteneur n'est pas déclaré comme `const`, il est possible de le modifier, lui ajouter ou lui supprimer des éléments. Voyez par vous-mêmes plusieurs exemples ci-dessous.

18.3.3. Cette clé est unique !

Abordons maintenant un point important de `std::unordered_map`, mais que j'ai passé sous silence. Il s'agit de **l'unicité de la clé**. Il ne peut y avoir qu'une seule valeur associée à une seule clé. Dit autrement, on ne peut pas assigner plusieurs valeurs à une même clé.

Quand on ajoute une nouvelle paire, il ne se passe rien de visible dans le cas où la clé existe déjà. C'est un peu gênant pour signaler l'erreur. Heureusement, la fonction `insert` retourne un `std::pair` contenant deux informations qui nous intéressent.

— L'élément `first` est un itérateur sur l'élément inséré si la clé n'existait pas au préalable, ou sur l'élément correspondant à la clé si elle existe déjà. Cet itérateur est lui-même un `std::pair`, donc on accède à la clé avec `first` et à la valeur avec `second`.
— L'élément `second` est un booléen indiquant si l'élément a été inséré (`true`) ou non (`false`).

Dans le cas où vous souhaitez ajouter une valeur si elle n'existe pas, ou bien mettre à jour la valeur associée à une clé si celle-ci existe, utilisez la fonction `insert_or_assign`, qui fonctionne presque pareil. Elle renvoie une paire semblable, avec toujours le booléen signalant l'insertion (`true`) ou la mise à jour (`false`).

18.3.4. Cherchez un élément

Comme l'accès à un élément par les crochets permet également d'ajouter une nouvelle paire clé / valeur, l'utiliser peut parfois être délicat. Voyez par vous-mêmes l'exemple suivant.

Il existe heureusement une fonction `find`, qui attend une clé et qui renvoie un itérateur sur l'élément concerné si la clé existe, ou bien un itérateur sur la fin de la collection sinon.

18.3.5. Un exemple concret

Grâce à ce que nous savons, nous pouvons faire un petit programme qui compte les occurrences de chaque mot au sein d'une phrase. Il suffit d'avoir un tableau associatif avec `std::string` pour clé et `int` pour valeur.

18.4. std : :unordered_set — Représenter un ensemble

Le deuxième conteneur se nomme `std::unordered_set`. C'est la représentation en C++ d'un ensemble au sens mathématique du terme, c'est-à-dire un ensemble d'éléments du même type, où chacun n'est présent qu'une seule et unique fois. C'est un peu comme une table associative dans laquelle il n'y aurait que des clés et pas de valeur.

> *i* **Fichier à inclure**
>
> Pour utiliser ce conteneur, il faut inclure le fichier d'en-tête `<unordered_set>`.

18.4.1. Inscription sur Zeste de Savoir

Sur le présent site, vous pouvez vous inscrire, mais il faut que votre pseudo soit unique. Vous ne pouvez donc pas réutiliser le pseudo de quelqu'un d'autre. On pourrait donc imaginer stocker les pseudos dans un `std::unordered_set`. Vous allez voir que l'utilisation de ce conteneur ressemble fortement à celle de `std::unordered_map`.

```
```

Comme vous le voyez, l'utilisation est très proche. La différence se trouve simplement dans l'absence de valeurs liées aux clés. Pour choisir entre les deux conteneurs, il faut simplement **réfléchir à ce que vous voulez stocker**. Un ensemble de clés uniques ? Choisissez `std::unordered_set`. Si vous devez associer à ces clés des valeurs, alors partez sur `std::unordered_map`.

18.5. Un peu d'ordre, voyons !

Ces deux conteneurs sont très utiles, mais comme vous l'avez remarqué, ils ne sont pas ordonnés. Si, par exemple, vous avez un `std::unordered_set` de chaînes de caractères, vous verrez qu'elles ne sont pas affichées dans l'ordre alphabétique.

```
```

```
```

En interne, les clés sont ordonnées et triées, mais selon **leur « hash »**. Non, je n'ai pas fait de faute de français. Un *hash* est le résultat d'une fonction de hachage. En très bref, c'est une fonction qui transforme une donnée quelconque en une donnée de taille fixée. Pour notre conteneur, cela signifie qu'il ne va plus travailler avec des chaînes de caractères ou autres clés potentiellement grosses, mais avec **une version de taille fixe et plus petite**, ce qui est **beaucoup plus rapide**.

Il existe une version alternative des conteneurs `std::unordered_map` et `std::unordered_set`, qui n'ordonne pas les éléments selon le hash des clés mais **directement avec les clés**. Ces nouveaux conteneurs se nomment, respectivement, `std::map` et `std::set`.

> **i Fichier à inclure**
>
> Pour les utiliser, il faut respectivement inclure les fichiers `<map>` et `<set>`.

En remplaçant `std::unordered_set` par sa version triée par clé dans l'exemple précédent, on obtient bien un affichage par ordre alphabétique.

```
```

> ? Du coup, la seule différence, c'est une question d'affichage et d'ordre ?

La structure interne et les algorithmes employés pour ajouter, supprimer ou chercher des éléments ne sont pas les mêmes. De manière générale, les versions non ordonnées sont plus rapides d'utilisation et ce sont celles qui sont utilisées dans 90% des cas. Les 10% restants se répartissent en deux points.

— On veut **conserver l'ordre des clés**.
— On a vraiment **énormément de données** (de l'ordre du million d'éléments), auquel cas la version non ordonnée est moins rapide que la version ordonnée, de part leur structure interne.

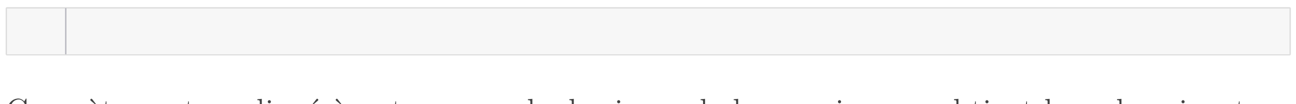
Leur utilisation est très semblable à leurs cousins non ordonnés, vous n'aurez donc aucun de mal à les manipuler.

18.6. Une longue énumération

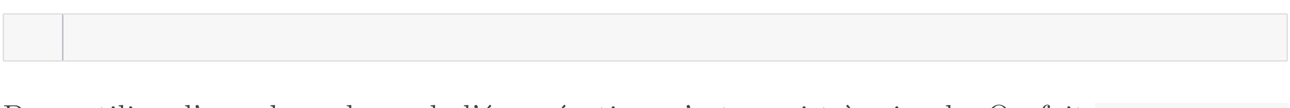
Terminons ce chapitre avec les énumérations. Leur but est de permettre une **liste de valeurs fortement typées et fixe**. Fortement typées car les valeurs sont d'un type bien précis et défini. Fixe car le nombre d'éléments est fixé et immuable.

On s'en sert typiquement pour **limiter le champs des possibilités** pour un type. Ainsi, au lieu de stocker le jour de la semaine sous forme de `std::string`, ce qui n'offre pas de protection si l'utilisateur rentre n'importe quoi, on va utiliser une énumération. Ainsi, celui-ci n'a pas d'autres choix que d'utiliser une des valeurs que nous avons nous-mêmes définis. C'est beaucoup plus souple que de devoir faire plein de vérifications sur une chaîne de caractères.

Pour déclarer une énumération, c'est très simple.



Concrètement appliqué à notre exemple des jours de la semaine, on obtient le code suivant.



Pour utiliser l'une des valeurs de l'énumération, c'est aussi très simple. On fait `Nom_enumeration::Valeur` (notez les deux-points `::`, comme dans `std::`). Examinez le code suivant qui montre un cas d'utilisation.

Ainsi, si vous avez une liste de choix de valeurs qui est limitée, **une énumération peut être la bonne solution**. Les mois de l'année, les couleurs d'un jeu de carte, les directions d'une boussole sont de bons cas d'utilisation. En plus de moins se casser la tête pour la vérification des valeurs, vous créez un nouveau type, clairement défini ; en effet, `JourSemaine` est bien plus parlant qu'une `std::string`. Le code est rendu **plus clair** et c'est une très bonne pratique.

Passage par recopie

Une énumération est très légère car elle utilise en interne des entiers. C'est pour cette raison qu'il est inutile de la passer par référence.

18.6.1. En résumé

- Les structures sont des agrégats de données permettant de lier et de manipuler plus aisément des données liées entre elles.
- Les tuples sont des collections hétérogènes et fixées, qui permettent de lier des données entre elles sans créer pour autant de nouveau type.
- Grâce aux tuples et aux structures, nous sommes en mesure de retourner plusieurs variables d'un coup.
- Les `std::unordered_map` sont des tables associatives, qui permettent d'associer une clé de type quelconque à une valeur de type quelconque.
- Les `std::unordered_set` sont des ensembles au sens mathématique du terme et permettent de stocker un ensemble de clés uniques.
- Il existe des versions triées par clé de ces conteneurs, `std::map` et `std::set`, mais qui sont beaucoup plus lentes.
- Une énumération permettent de définir un type avec un ensemble de valeurs fixe et clairement défini.

Contenu masqué

Contenu masqué n°43 : Correction

```
```

Retourner au texte.

Contenu masqué n°44 : Correction

Retourner au texte.

19. Reprendriez-vous un peu de sucre syntaxique ?

Le sucre se trouve partout aujourd'hui. Dans les gâteaux, dans les biscuits, dans les boissons et même dans notre code ! Sauf que dans ce dernier cas, on parle de **sucre syntaxique**. Ce terme, inventé par Peter J. Landin , un informaticien britannique, désigne des syntaxes plus lisibles ou plus faciles à écrire qu'on utilise pour se faciliter la vie.

Le but de ce chapitre va donc être d'**utiliser** et même d'**écrire du sucre syntaxique**.

19.1. Ce type est trop long !

Vous vous souvenez du chapitre sur les flux ? Nous y avons découvert comment traiter des chaînes de caractères comme des flux, avec les types `std::istringstream` et `std::ostringstream`. Mais il faut avouer que c'est assez long à écrire et fatiguant à la longue.

Heureusement, en C++, on peut définir des **alias de types**, des synonymes, qui permettent d'éviter l'utilisation de types longs à écrire ou bien de types composés. Cela ce fait très simplement, en utilisant le mot-clef `using`.

```

```

Même la bibliothèque standard utilise le concept. Ainsi, `std::string` est lui-même l'alias d'un type `std::basic_string<char>`. Notez que c'est quand même beaucoup plus agréable à écrire.

19.2. Décomposons tout ça

Dans le chapitre précédent, nous avons souligné la lourdeur et le manque d'expressivité de l'accès aux éléments d'un tuple. Il est vrai qu'une expression comme `std::get<2>(tuple)` ne nous donne guère d'information sur le sens de ce que nous manipulons. Heureusement, il existe des syntaxes alternatives, que nous allons voir.

19.2.1. En pleine décomposition

Une des nouveautés de C++17, ce sont les décompositions (en anglais « *structured bindings* »). Ce concept est très simple. Son but est de déclarer et initialiser plusieurs variables d'un coup, depuis un tuple ou une structure. Et comme un code vaut mille mots, regardez par vous-mêmes. J'ai repris un exemple du chapitre précédent, en l'adaptant.

Notez comment le code se voit simplifié et clarifié. Nous avons remplacé une ligne aussi peu claire que `std::get<0>(calculs)` par `cosinus`. D'un coup d'œil, vous voyez ce que fait la variable et ce que nous manipulons. Le code est plus lisible et donc de meilleure qualité. N'oubliez pas qu'un code est bien plus souvent lu qu'écrit.

On peut bien entendu utiliser des références, si l'on veut que des modifications sur les variables se répercutent sur le tuple lié. L'exemple suivant est tiré d'un billet de @germinolegrand.

Enfin, ce que nous venons de voir est tout à fait applicable aux structures. L'exemple suivant, bien que bidon, vous le montre.

Une décomposition peut déclarer exactement autant de variables que d'éléments composant la structure ou le tuple, ni plus, ni moins. Même si nous ne voulons récupérer que certaines variables, il faut tout de même créer des variables inutiles qui ne seront plus utilisées par la suite.

19.2.2. Et avant, on faisait comment ?

Les décompositions sont apparues en C++17, mais avant, il existait une autre façon de faire, qui existe toujours soit dit en passant. Il s'agit de `std::tie`. Le principe est identique, puisque on veut assigner à des variables les différents éléments d'un tuple ou d'une structure. Sauf qu'ici, les variables doivent être déclarées au préalable.

Cette solution présente a comme inconvénient majeur que les variables doivent être déclarées à l'avance et donc qu'**on ne peut pas profiter de l'inférence de type**. Par contre, l'avantage qu'elle a sur la décomposition, c'est qu'elle permet **d'ignorer certaines valeurs** qui ne nous intéressent pas et que nous ne voulons pas récupérer, grâce à `std::ignore`.

Un autre avantage, c'est de pouvoir réutiliser une même variable plusieurs fois, alors que la décomposition C++17 ne nous le permet pas.

19.3. La surcharge d'opérateurs

Lorsque l'on crée un type, il arrive que l'on souhaite réaliser certaines opérations communes sur ce type, comme l'injection dans un flux (pour de l'affichage par exemple), la réalisation d'opérations arithmétiques (addition, soustraction, etc) ou encore d'opérations de comparaison.

19.3.1. Le problème...

Imaginons que nous voulions créer un type pour manipuler des fractions. On aimerait pouvoir les additionner, les soustraire, etc, en bref, faire des opérations dessus. Sauf qu'on ne peut pas se contenter de faire un simple `+` ou `-`, comme c'est le cas avec des `int` ou des `double`.

On doit alors écrire des fonctions pour faire ça. Mais si on les mélange, ça peut très vite devenir indigeste à lire.

On peut également parler de l'affichage qui est galère et bien d'autres choses encore. Il faut donc trouver une solution à cet énorme problème de lisibilité. L'idéal, ça serait de pouvoir utiliser les opérateurs avec nos types personnels. Et heureusement, C++ ne nous laisse pas seuls face à ce problème !

19.3.2. …et la solution

Dans le chapitre introduisant les fonctions, nous avons vu que l'on peut surcharger une fonction pour la faire fonctionner avec des paramètres de types différents. Eh bien, bonne nouvelle, **les opérateurs sont des fonctions comme les autres**.

Autrement dit, tous les opérateurs, aussi bien arithmétiques (`+`, `*`, etc), de comparaison (`<`, `>`, `==`, etc), ou de flux (`<`, `>`), sont appelables et surchargeables. Simplement, ce sont des fonctions avec une syntaxe spéciale faite pour rendre le code plus lisible.

Ainsi, on va pouvoir surcharger nos opérateurs pour nos types personnels et ainsi éliminer le problème d'expressivité du code qu'on a soulevé précédemment.

19.3.3. Cas concret — Les fractions

20. Les opérateurs arithmétiques

Nous allons aborder ici les opérateurs `+`, `-`, `*` et `/`, qui nous permettront de réaliser, respectivement, des additions, des soustractions, des multiplications et des divisions de fractions. Voyons déjà à quoi ressemblent, de manière générale, les opérateurs en question.

Dans notre cas, ils se traduisent ainsi. Notez que les prototypes de `+` et de `-` ressemblent beaucoup aux fonctions `additionner` et `soustraire` que nous avons écrites.

> **Remarque**
>
> Le résultat de `64/64` peut vous surprendre, mais il est tout à fait normal, parce qu'on ne simplifie pas la fraction.

Entraînez-vous en implémentant aussi la multiplication et la division. Le code est très proche de celui qu'on vient d'écrire.

20.0.0.1. Les opérateurs de comparaison

Continuons en essayant maintenant de comparer des fractions entre elles, pour savoir notamment si deux fractions sont égales ou non, comparer la grandeur de deux fractions, etc. C'est le même principe, les prototypes peuvent se déduire assez facilement.

Et si vous êtes effrayés à l'idée d'implémenter autant d'opérateurs différents, rappelez-vous de la logique booléenne. Vous verrez, ça réduit énormément la quantité de code.

Aviez-vous pensé à l'astuce consistant à réutiliser le code déjà écrit pour simplifier celui à écrire ? Si oui, c'est que vous commencez à acquérir les bons réflexes de développeur.

20.0.0.2. Les opérateurs de manipulation des flux

Il nous reste le problème d'affichage de notre type `Fraction`. Pour l'instant, nous sommes obligés d'aller chercher le numérateur et le dénominateur nous-mêmes. Si notre type contient plein d'informations, toutes aller les récupérer devient vite pénible.

Les opérateurs de manipulation des flux `<` et `>` sont, fort heureusement, surchargeables. Voyez leur prototype.

Ce qui peut surprendre, c'est la fait que l'on renvoie une référence sur un flux à chaque fois. En fait, c'est le flux que l'on reçoit en argument qui est renvoyé. Ceci a pour but d'enchaîner des appels à l'opérateur pour pouvoir écrire du code tel que celui-ci.

Il sera interprété ensuite comme une suite d'appel à l'opérateur `<` par le compilateur.

> **Type retourné**
>
> Notez que les types retournés sont `std::ostream` et `std::istream`. Ce sont des **versions génériques de flux de sortie et d'entrée**. Ainsi, en renvoyant ces types, on s'assure que notre type peut être utilisé avec des flux vers l'entrée / sortie standards, vers des fichiers, des chaînes de caractères, etc.

Dans notre cas, celui qui nous intéresse le plus, c'est `operator<`, afin de pouvoir afficher nos fractions sur la sortie standard.

Si vous le souhaitez, vous pouvez vous amuser à implémenter l'opérateur `>`, pour, par exemple, être capable de créer des `Fraction` depuis des chaînes de caractères ou en lisant un fichier.

20.0.0.3. Plus et moins unaires

Il existe encore beaucoup d'opérateurs, nous n'allons donc pas pouvoir tous les aborder dans ce chapitre. J'aimerais néanmoins vous parler du plus et du moins unaires, qui permettent de faire des opérations comme `+variable` ou `-variable`, exactement comme des entiers (entre autres).

Le modèle est très simple, le voici ci-dessous pour les deux opérateurs.

Dans notre code, cela nous permet donc de créer, notamment, des fractions négatives.

20.0.0.4. Aller plus loin

Pour pratiquer, je vous propose de finir ce petit projet de fractions. Voici quelques idées en vrac.

- Il n'y a aucune gestion des erreurs. Comment y remédier ?
- il n'y a pas de tests unitaires.
- Renvoyer l'opposé d'une fraction.
- Inverser une fraction.
- Rendre une fraction irréductible. Pour cela, il faut diviser le numérateur et le dénominateur par leur PGCD.

20.0.1. Et la bibliothèque standard ?

Elle utilise à foison la surcharge d'opérateurs. Et vous en avez utilisé énormément sans le savoir. Examinons quelques cas.

20.0.1.1. Concaténation de chaînes

Quand vous concaténez des chaînes de caractères, vous faites en réalité appel à `operator+` appliqué au type `std::string`. L'utilisation de cet opérateur est parfois décrié par certains, qui le jugent peu adapté. Reste que, celui-ci ou un autre, le code est raccourci et plus agréable à lire.

20.0.1.2. Accès à un élément

Dans l'exemple précédent, vous pouvez remplacer `std::vector` par `std::array` ou `std::map`, le principe est toujours le même. Derrière, c'est en effet `operator[]` qui est appelé. Celui-ci attend un entier comme argument, ce qui ne doit pas vous surprendre, puisque vous êtes habitués à la manipulation de tableaux maintenant.

20.0.1.3. Définition de littéraux `std::string`

Vous vous demandez où est la surcharge dans le code précédent ? Allez, je vous aide, c'est le `s` en fin de chaîne. Eh oui, même quelque chose d'aussi simple est en fait une surcharge de `operator""`. Ce petit nouveau, arrivé avec C++11, permet de définir des suffixes, qui s'ajoutent après des littéraux, afin d'en changer le type.

Dans le cas exposé ci-dessus, le suffixe `s` permet de transformer le type du littéral `"Hello, chaine C++"`, qui était un type hérité du C, en un `std::string`. Sans cette surcharge, il aurait fallu écrire `std::string { "Hello, chaine C++" }`, ce qui est plus long et moins lisible.

20.0.2. Exercice — Calculs avec des durées

Pratiquons pour mieux retenir avec un petit exercice, dans lequel on va implémenter quelques opérations pour un type `Duree` qui, comme son nom l'indique, représente une durée. Voici déjà comment on définit une durée dans notre code.

Comme vous voyez, on stocke les durées sous forme de secondes. L'exercice consiste à implémenter une liste d'opérations.

- Créer une durée à partir de secondes, minutes (optionnel), heures (optionnel).
- Renvoyer la durée sous forme de secondes.
- Renvoyer la durée sous forme de tuple {minutes, secondes}. Le signe de la durée sera porté par les minutes.
- Renvoyer la durée sous forme de tuple {heures, minutes, secondes}. Le signe de la durée sera porté par les heures.
- Additionner des durées.
- Renvoyer l'opposé d'une durée.
- Soustraire des durées.
- Comparer des durées.
- Injecter une durée dans un flux sortant, au format `H:mm'ss"`. Le signe de la durée sera mis devant les heures.

Définition d'une durée

Ici, on considère une durée comme un intervalle de temps entre deux événements. Par conséquent, une durée peut être négative ; les opérations comme prendre l'opposé d'une durée ont donc un sens.

20.0.2.1. Indice

Voici, pour vous aider, les prototypes des fonctions à implémenter.

Prototypes

20.0.2.2. Corrigé

Voici le corrigé complet avec les implémentations.

Corrigé complet

20.0.3. Sur le bon usage de la surcharge

La surcharge d'opérateurs est un outil puissant, il faut donc faire attention à ne pas l'utiliser n'importe comment. Notamment, C++ ne vérifie pas ce que vous mettez dans votre implémentation. Vous pourriez alors surcharger l'opérateur `/` pour effectuer une addition. Dans ce cas, vous violeriez la **sémantique** des opérateurs, c'est-à-dire que vous leur feriez faire autre chose que ce pour quoi ils sont prévus normalement. C'est une **très mauvaise pratique**, à bannir absolument.

L'autre point, c'est que, même si vous pouvez surcharger des opérateurs, **cela n'a pas toujours de sens de le faire**. Ainsi, additionner deux personnes ou deux comptes en banque **n'a aucun sens et n'est pas correct**. Nous reviendrons sur cet aspect plus tard dans le cours. Retenez que ce n'est pas parce que vous pouvez que vous devez.

20.0.4. En résumé

— On peut définir des alias de type grâce au mot-clé `using`.
— On peut simplifier l'utilisation de `std::tuple` et même de structures grâce aux décompositions.
— La surcharge d'opérateurs permet d'augmenter la lisibilité et la qualité de notre code.
— Il est possible de surcharger de nombreux opérateurs.

— Elle doit être utilisée à bon escient, en ne faisant pas faire aux opérateurs des opérations qui violent leur sémantique.
— La surcharge d'opérateurs est un bonus qui, s'il est appréciable, ne doit pas pour autant être constamment et systématiquement utilisé.

Contenu masqué

Contenu masqué n°45 : Prototypes

Retourner au texte.

Contenu masqué n°46 : Corrigé complet

Retourner au texte.

21. [T.P] Un gestionnaire de discographie

Depuis le début de ce cours, nous avons eu l'occasion de découvrir de nombreux concepts, que l'on a pratiqués lors des exercices et des mini-T.P.

Dans ce chapitre, nous allons pour la première fois attaquer un gros T.P, dont le but va être d'utiliser toutes les notions acquises jusqu'à présent pour produire un programme complet et fonctionnel. C'est votre baptême du feu, en quelque sorte. Mais ne vous inquiétez pas, on va y aller pas à pas.

21.1. L'énoncé

Lors de ce T.P, vous allez créer un gestionnaire de discographie en ligne de commande. C'est un logiciel qui va permettre d'ajouter des morceaux avec album et artiste à votre discographie. Il donnera alors la possibilité d'afficher votre discographie de plusieurs manières différentes, ainsi que de l'enregistrer et de la charger à partir de fichiers.

Toutes les actions devront être faisables par l'utilisateur en ligne de commande. Il aura une petite invite de commande attendant qu'il rentre ses ordres.

```

```

21.1.1. Ajout de morceaux

Pour ajouter un morceau à la discographie, l'utilisateur le fera simplement en tapant une commande de cette forme :

```

```

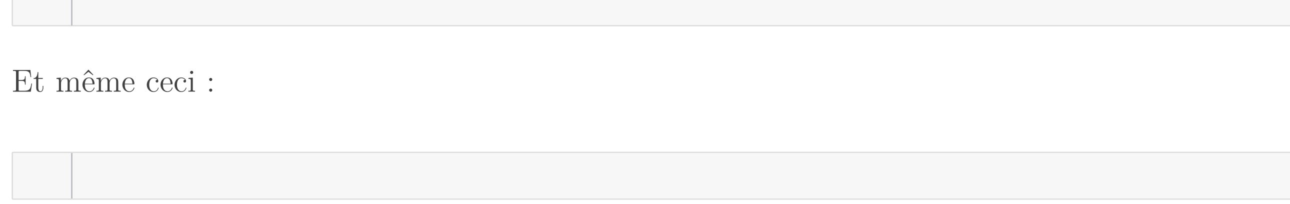
Les informations sont optionnelles. Par exemple, l'utilisateur doit pouvoir taper ça :

```

```

Ou ça :

```

```

Ou encore ceci :



Et même ceci :

Lorsque des informations sont omises, la discographie enregistrera le morceau avec les mentions «Morceau inconnu», «Album inconnu» ou «Artiste inconnu».

21.1.2. Affichage de la discographie

L'utilisateur doit pouvoir afficher sa discographie de trois manières différentes, que l'on va illustrer par un exemple. Les commandes d'affichage commenceront toutes par le même mot.

Prenons pour exemple la discographie suivante.

Morceau	Album	Artiste
Bad	Bad	Michael Jackson
Bloody Well Right	Crime of the Century	Supertramp
It's Raining Again	...Famous Last Words...	Supertramp
Buffalo Soldier	Confrontation	Bob Marley and the Wailers
The Way You Make Me Feel	Bad	Michael Jackson

21.1.2.1. Affichage par ordre alphabétique des morceaux

L'utilisateur pourra taper `> afficher morceaux`, auquel cas le programme devra lui afficher la liste des morceaux par ordre alphabétique.

21.1.2.2. Affichage par album

Si l'utilisateur tape `> afficher albums`, le programme devra lui répondre avec la liste des albums, avec une sous-liste des morceaux de ces albums, le tout étant trié par ordre alphabétique.

21.1.2.3. Affichage par artiste

C'est le même principe : si l'utilisateur tape `> afficher artistes`, le programme répondra une liste des auteurs, avec une sous-liste d'albums contenant elle-même une sous-liste de morceaux.

Bien sûr, si vous voulez adapter un peu l'affichage pour qu'il vous plaise plus, vous pouvez. Après tout, l'un des intérêts de savoir programmer, c'est de pouvoir faire des programmes qui nous plaisent !

21.1.3. Enregistrement et chargement d'une discographie

L'utilisateur doit pouvoir enregistrer sa discographie sur le disque, pour pouvoir la sauvegarder lorsqu'il arrête le programme. Il doit aussi pouvoir la charger depuis un fichier. Ces deux fonctionnalités s'utiliseront à travers deux commandes.

21.1.4. Quitter le programme

Lorsqu'il souhaite quitter le programme, l'utilisateur aura simplement à taper la commande qui suit.

21.1.5. Gestion des erreurs

Le programme devra être robuste aux erreurs de saisie : si l'utilisateur tape une commande invalide, le programme ne devra pas crasher ou avoir un comportement bizarre. Il doit ignorer la commande et signaler à l'utilisateur que la commande est invalide.

Le programme devra aussi être capable de gérer les erreurs dues à l'impossibilité d'ouvrir un fichier, que ce soit en lecture ou en écriture.

21.1.6. Dernières remarques

Ce T.P est votre première confrontation à la conception logicielle. La difficulté ne réside plus seulement dans la compréhension et l'utilisation du langage, mais aussi dans la conception : il va vous falloir réfléchir à comment organiser votre code.

L'objectif est que vous réfléchissiez à une manière d'avoir un code propre, robuste, agréable à lire, ouvert aux évolutions, et bien sûr fonctionnel.

C'est une tâche difficile, mais rassurez-vous : comme c'est votre première fois, nous vous proposons un guide pour pouvoir avancer dans le T.P. Prenez cependant le temps de réfléchir et cassez-vous la tête sur le T.P sans utiliser le guide. Cela vous confrontera à la difficulté et vous fera progresser. Ensuite, vous pourrez lire le guide pour vous aiguiller vers une solution. Et comme d'habitude, on vous proposera un corrigé complet à la fin.

21.2. Guide de résolution

21.2.1. Analyse des besoins

Pour concevoir un programme efficacement, il faut analyser les besoins auxquels il doit répondre — autrement dit les tâches qu'il doit être capable de réaliser — pour arriver à des problèmes plus atomiques, auxquels on peut répondre directement par un morceau de code, une fonction, une structure de données…

Ici, les besoins sont définis par l'énoncé ; on va donc partir de là et les découper en petits problèmes.

21.2.1.1. Un interpréteur de commande

Déjà, l'utilisateur donne ses instructions au moyen d'une ligne de commande. Notre logiciel doit donc être capable de lire et comprendre les commandes de l'utilisateur, ainsi que de détecter les commandes invalides.

Il y a plusieurs types de commande : les commandes d'ajout de morceau, les commandes de sauvegarde/chargement de la bibliographie, et bien sûr les commandes d'affichage. Celles-ci se décomposent ensuite en plusieurs modes d'affichage. On a donc deux tâches distinctes.

- Lire et interpréter les commandes.
- Exécuter les commandes.

On va avoir une fonction d'exécution pour chaque type de commande, et des fonctions qui se chargeront de lire les commandes et d'appeler la fonction d'exécution adéquate ; ce seront nos fonctions d'interprétation. Nous reviendrons un peu plus tard sur l'implémentation, mais on a déjà une idée du schéma de fonctionnement global.

21.2.1.2. Le besoin de stocker la discographie

Évidemment, il nous faut un moyen de mémoriser la discographie, que ce soit pendant l'exécution du programme (stockage en mémoire de travail dans une variable) ou sur le long terme (stockage sur disque dans un fichier). On dégage donc ici deux besoins techniques :

— Le premier est de pouvoir représenter la bibliographie dans notre programme ; en C++, cela revient à peu de choses près à définir un type adapté. C'est par cet aspect que nous rentrerons en premier dans les détails techniques ; le choix de commencer par là est pertinent car cette représentation est centrale et sera omniprésente dans le reste du code.
— Le second est de réussir à sauvegarder et charger cette représentation dans un fichier ; on doit entre autres choisir une représentation de la bibliothèque sous un format texte.

21.2.1.3. Le besoin de gérer les erreurs d'entrée

L'énoncé explique qu'il faut gérer les erreurs d'entrée - et ce de manière *user friendly*. Plus précisément, il faut que nos fonctions d'interprétation détectent les erreurs. Elles doivent aussi pouvoir les remonter. Pour cela, on peut tout simplement utiliser le système d'exceptions. On y reviendra plus en détails un peu plus loin.

21.2.2. Attaquons le code !

On peut maintenant aborder la conception plus précise des différents points abordés précédemment, ainsi que l'implémentation. Comme on l'a dit, on va commencer par un point central : la représentation de la discographie.

21.2.2.1. Représentation de la discographie

Une discographie peut simplement être vue comme une collection de morceaux. D'après l'énoncé, les informations dont on a besoin pour un morceau sont son nom, son album et son artiste. Ainsi :

— Un artiste a comme unique donnée son nom.
— Un album a lui aussi un nom.
— Un morceau a pour données importantes son nom, ainsi que l'album et l'artiste correspondants.
— Une discographie est une collection de morceaux, de taille variable évidemment.

Ces constats étant fait, cela devrait vous aider à trouver des structures de données adaptées.

Structures de données

21.2.2.2. Fonctions de base

Une fois nos structures de données définies, il est commun de créer des fonctions de base qui vont nous permettre de les manipuler de manière agréable. C'est typiquement le moment de créer des surcharges pour certains opérateurs pour nos types nouvellement créés. Notamment, on aimerait pouvoir afficher facilement nos différentes structures, ainsi que les lire et les écrire dans un fichier. On va donc avoir besoin de surcharger les opérateurs de flux.

Je vous suggère dans un premier temps d'essayer de trouver les prototypes de telles fonctions, puis d'essayer de les implémenter.

Prototypes

Implémentations

Comme nous sommes des programmeurs consciencieux, nous voulons écrire des tests pour nous assurer que ce que nous avons déjà écrit fonctionne bien. En plus de vérifier qu'une entrée valide est acceptée, nous devons aussi penser aux cas limites ainsi qu'aux erreurs, pour mesurer la résistance et la bonne conception de notre programme.

Par exemple, `"Frica | Frica | Carla's Dreams"` et `"Levels         |   Levels    | Avicii"` sont des entrées valides et complètes, même si dans la seconde il y a beaucoup d'espaces. De même, `"Subeme la radio | | Enrique Iglesias"` et `"Duel of the fates | |"` aussi, quand bien même certaines informations sont manquantes.

Tests unitaires

21.2.2.3. Le stockage dans un fichier

Maintenant que l'on a construit nos structures de base pour pouvoir représenter une bibliographie en mémoire, on peut créer des fonctions pour sauvegarder et charger une discographie dans un fichier.

Implémentation

21.2.2.4. L'affichage de la discographie

Autre point qu'on a soulevé, l'affichage. Nous voudrions pouvoir afficher la discographie par albums, par artistes ou par titres. Plus précisément, on remarque que chaque mode d'affichage, il y a en réalité deux composantes distinctes : la manière d'afficher à proprement parler, mais aussi l'ordre dans lequel sont affichés les morceaux, c'est-à-dire le tri de la discographie. Nous

dégageons ainsi six fonctions à implémenter : une de tri et une d'affichage pour chaque mode d'affichage.

Prototypes

Implémentation

> La manière dont j'ai remarqué le fait qu'il y avait deux parties bien distinctes à envisager - et donc à traiter dans des fonctions séparées - est très intéressante, donc j'aimerais vous l'expliquer.
> Au départ, le tri se faisait directement dans les fonctions d'affichage. Or des fonctions d'affichage, vu leurs rôles, ne devraient pas avoir à modifier la discographie ; en bon programmeur qui respecte la *const-correctness*, je souhaitais donc que ces trois fonctions prennent en paramètres une `Discographie const&` et non pas une `Discographie&`. Mais le tri modifie la discographie, donc je ne pouvais pas. C'est comme ça que j'ai remarqué que la logique d'affichage comprenait en fait deux sous-tâches indépendantes, à traiter séparément.
> Ainsi, on est dans un cas où faire attention à la *const-correctness* permet d'améliorer la conception du code.

Pour lier tout ça, on peut créer une quatrième fonction qui lancera les fonctions du mode d'affichage demandé. Comme nos possibilités d'affichage sont limitées, on peut les regrouper dans une énumération.

Prototypes

Implémentation

21.2.2.5. Le système de commandes

Il nous reste encore à interagir avec l'utilisateur, à l'aide du système de commandes. On a fait ressortir deux problèmes principaux, l'interprétation et l'exécution des commandes. On gardera en tête le fait qu'il faut gérer les erreurs.

Prototypes

Essayez maintenant d'implémenter de telles fonctions.

Implémentations

21.2.2.6. Finalisation

Maintenant que toutes nos fonctionnalités sont programmées, il suffit de tout intégrer en écrivant une fonction `main`, qui aura pour principal mission d'initialiser le programme (en lançant d'abord les tests) et de faire tourner une boucle d'invite de commandes jusqu'à ce que l'utilisateur décide de quitter.

Fonction principale

Et voilà ! On a construit pas à pas toutes les briques de notre programme, jusqu'à avoir tout le code pour le faire fonctionner. La démarche à retenir est la suivante : il faut réfléchir à une manière de découper notre problème initial — ici « faire une discographie » — en sous-problèmes, par exemple « lire et comprendre une commande », qui eux-mêmes seront découpés en sous-problèmes, jusqu'à arriver à des briques de base simples à réaliser — par exemple « créer un type représentant les morceaux ».

21.3. Corrigé complet

Bon, j'espère que vos phases de recherches et d'avancées à l'aide du guide ont été fructueuses !

Que vous ayez réussi ou pas à avoir un résultat fonctionnel, l'important est que vous ayez bien réfléchi au T.P. C'est en vous confrontant à vos difficultés que vous progresserez.

Bref, trêve de bavardages, voici sans plus attendre une solution possible au T.P, issue du guide ci-dessus.

Corrigé complet

Une façon parmi d'autres

Ce code n'est pas LA solution, mais une parmi tant d'autres. Le vôtre est différent, mais vous êtes malgré tout arrivés à la solution ? Bravo, vous pouvez être fiers.

21.4. Conclusion et pistes d'améliorations

Voilà qui conclut ce TP, dans lequel vous avez pu vous confronter pour la première fois à la conception d'un programme complet, plus ambitieux que les petits exercices. Cela nous a permis de voir la démarche qu'il faut avoir en tête pour pouvoir créer du code de qualité, c'est-à-dire expressif et surtout évolutif.

Justement, cette évolutivité vous permet d'ajouter des fonctionnalités à votre programme à votre guise, et ce de manière efficace. Vous pouvez ajouter ce que vous voulez. Ci-dessous, voici quelques idées.

— Ajouter des options d'affichage comme l'affichage des morceaux d'un seul album ou artiste, grâce à des commandes du type `> afficher artiste Dave Brubeck` par exemple.
— Ajouter des options de tri des morceaux, par BPM par exemple.
— Avoir un fichier de configuration du logiciel permettant à l'utilisateur de choisir son invite de commandes, son séparateur dans les affichages et dans les commandes d'ajout, les puces à utiliser dans les listes affichées, etc. En bref, rendre le logiciel configurable. Un tel fichier pourrait ressembler à ça, pour les réglages donnant les mêmes affichages que ceux qu'on a codés en dur.

```

```

— Notre code n'est pas entièrement testé. Il ne tient qu'à vous d'écrire les tests manquants.

Les possibilités d'ajout sont infinies, vous pouvez faire ce que vous souhaitez.

N'hésitez pas, lorsque vous voulez ajouter des fonctionnalités, à continuer avec la même démarche : commencez par réfléchir, décomposez ce que vous voulez faire en petits problèmes, créez des types pratiques s'il le faut, etc.

> **i** Conseil d'ami
>
> Je vous conseille de lire le chapitre suivant avant de faire des ajouts au TP. En effet, vous serez alors capables de mieux organiser votre code, ce qui vous aidera à le faire évoluer pour ajouter des nouvelles fonctionnalités.

21.4.1. En résumé

— Vous vous êtes confrontés pour la première fois à la réalité de la conception d'un programme informatique.
— On a pu en tirer une démarche pour avoir un code lisible et évolutif.
— On a également pu voir l'importance de la réflexion si on veut obtenir un code de qualité.
— On a aussi vu comment les notions vues jusqu'à présent peuvent se combiner en pratique, pour en tirer le meilleur et ainsi obtenir cette qualité.
— Nous avons vu quelques pistes possibles d'amélioration, mais il ne tient qu'à vous d'ajouter les fonctionnalités que vous souhaitez, comme vous le souhaitez.

Contenu masqué

Contenu masqué n°47 : Structures de données

Un choix possible est le suivant : représenter les artistes, albums et morceaux par des structures simples, et la discographie par un alias, à l'aide de `using`.

Retourner au texte.

Contenu masqué n°48 : Prototypes

Retourner au texte.

Contenu masqué n°49 : Implémentations

Remarquez l'utilisation des flux de chaînes de caractères pour gagner en clarté et en élégance. Ils seront beaucoup utilisés dans ce T.P.

Vous pouvez aussi voir que l'on a défini une fonction en plus, la fonction `traitement_chaine` : c'est une fonction utilitaire créée pour éviter la répétition de code. Retourner au texte.

Contenu masqué n°50 : Tests unitaires

Retourner au texte.

Contenu masqué n°51 : Implémentation

```

```

Retourner au texte.

Contenu masqué n°52 : Prototypes

```

```

Retourner au texte.

Contenu masqué n°53 : Implémentation

```

```

Retourner au texte.

Contenu masqué n°54 : Prototypes

```

```

Retourner au texte.

Contenu masqué n°55 : Implémentation

```

```

On utilise ici une exception qui ne devrait en principe jamais être lancée, car nous ne tomberons jamais dans le `else`. Mais si, plus tard, on modifiait le programme pour ajouter une nouvelle forme d'affichage, sans mettre à jour cette partie du code, l'exception nous le rappellera. Retourner au texte.

Contenu masqué n°56 : Prototypes

Remarquez la création d'une énumération pour représenter les différents types de commande ; ce n'est pas nécessaire, c'est juste un type de base de plus pour pouvoir lire élégamment le type de commande dans la console. Retourner au texte.

Contenu masqué n°57 : Implémentations

Comme on l'a dit, on utilise les exceptions pour faire remonter les erreurs en cas de commande invalide. C'est un usage logique des exceptions, puisque les erreurs de commande de l'utilisateur sont indépendantes de la volonté du programmeur. Les exceptions sont donc toutes indiquées pour gérer ce genre de cas. Retourner au texte.

Contenu masqué n°58 : Fonction principale

Il ne faut pas oublier de gérer les erreurs remontées par notre programme. Ici ce n'est pas compliqué, il suffit d'expliquer l'erreur à l'utilisateur et de lui redemander une entrée. Retourner au texte.

Contenu masqué n°59 : Corrigé complet

Retourner au texte.

22. Découpons du code — Les fichiers

Lors du chapitre précédent, nous avons codé notre premier programme d'assez grande ampleur. Et avec l'ampleur que prend notre code, vient un problème qui était caché par la taille réduite de ce qu'on avait fait jusqu'alors : **l'organisation du code**.

En effet, jusqu'à maintenant, on mettait simplement tout notre code dans le fichier `main.cpp`. Le plus sophistiqué qu'on ait fait en termes d'organisation est d'utiliser les prototypes pour pouvoir reléguer les implémentations après la fonction `main` et ainsi alléger un peu la lecture du code.

Jusque-là ça suffisait, mais on voit déjà avec le TP que dès que le projet prend un peu d'ampleur, ça devient un sacré fouillis. Et encore, ça n'est rien comparé à certains gros projets. Par exemple, à l'heure où j'écris ces lignes, le code du noyau Linux comporte 17 millions de lignes !

C'est pourquoi on va maintenant apprendre **comment séparer son code en plusieurs fichiers**, ce qui va nous permettre de mieux l'organiser.

22.1. Le principe

Nous savons déjà séparer les prototypes de leur implémentation, ce qui est déjà une première étape vers une réorganisation. Mais ce n'est pas suffisant. Il faut maintenant séparer le code en plusieurs fichiers. Et il en existe deux types.

22.1.1. Le fichier d'en-tête

Les fichiers d'en-tête contiennent **les déclarations des types et fonctions que l'on souhaite créer**. Nous sommes habitués à les manipuler depuis le début de notre aventure avec C++. C'est grâce à eux que nous sommes en mesure d'utiliser la bibliothèque standard. Le concept n'est pas nouveau, on passe simplement de consommateur à créateur.

Les fichiers d'en-tête peuvent tout à fait **inclure d'autres fichiers d'en-tête**, tant standards que personnels. C'est le cas par exemple où notre fichier d'en-tête fait appel à des composants de la bibliothèque standard, comme, entre autres, `std::string` ou `std::vector`.

22.1.1.1. Sur Visual Studio

Ouvrez votre projet avec Visual Studio. Vous tombez sur un écran semblable au mien.

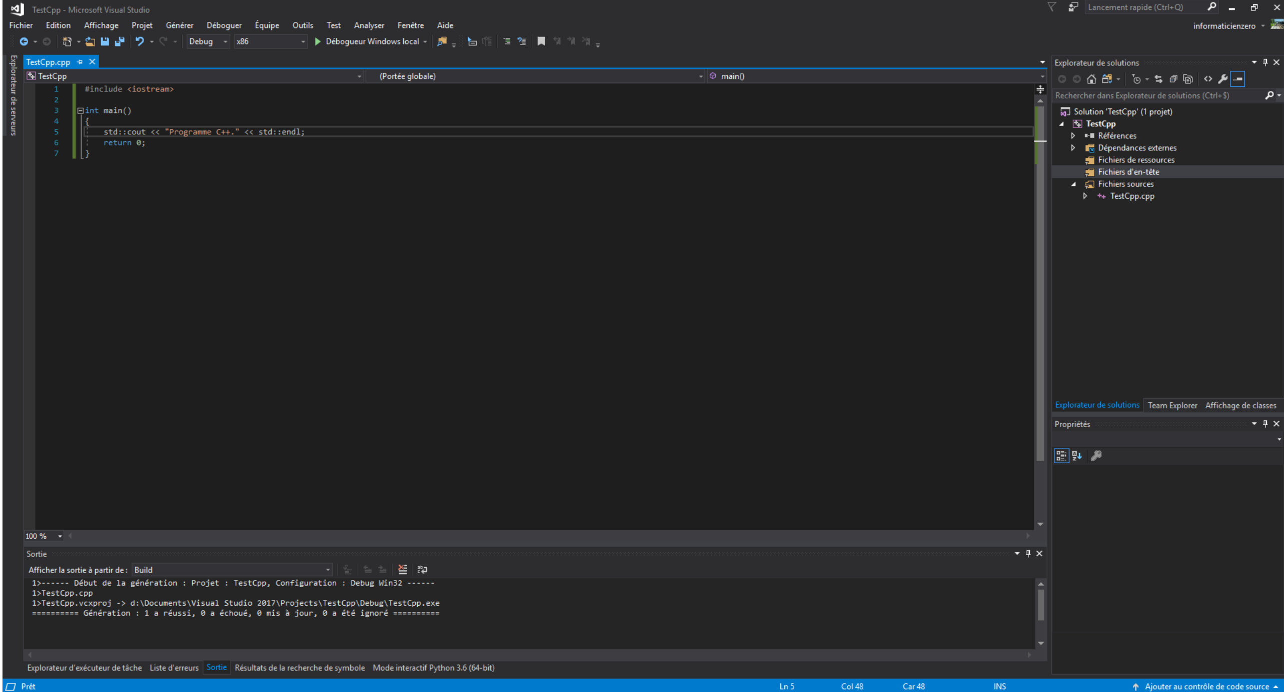


Figure 22.1. – Page principale de Visual Studio avant l'ajout d'un fichier d'en-tête au projet.

Faîtes un clic-droit sur le nom du projet (ici, *TestCpp*), puis cliquez sur `Ajouter -> Nouvel élément...`. Ce faisant, vous arrivez sur l'écran suivant.

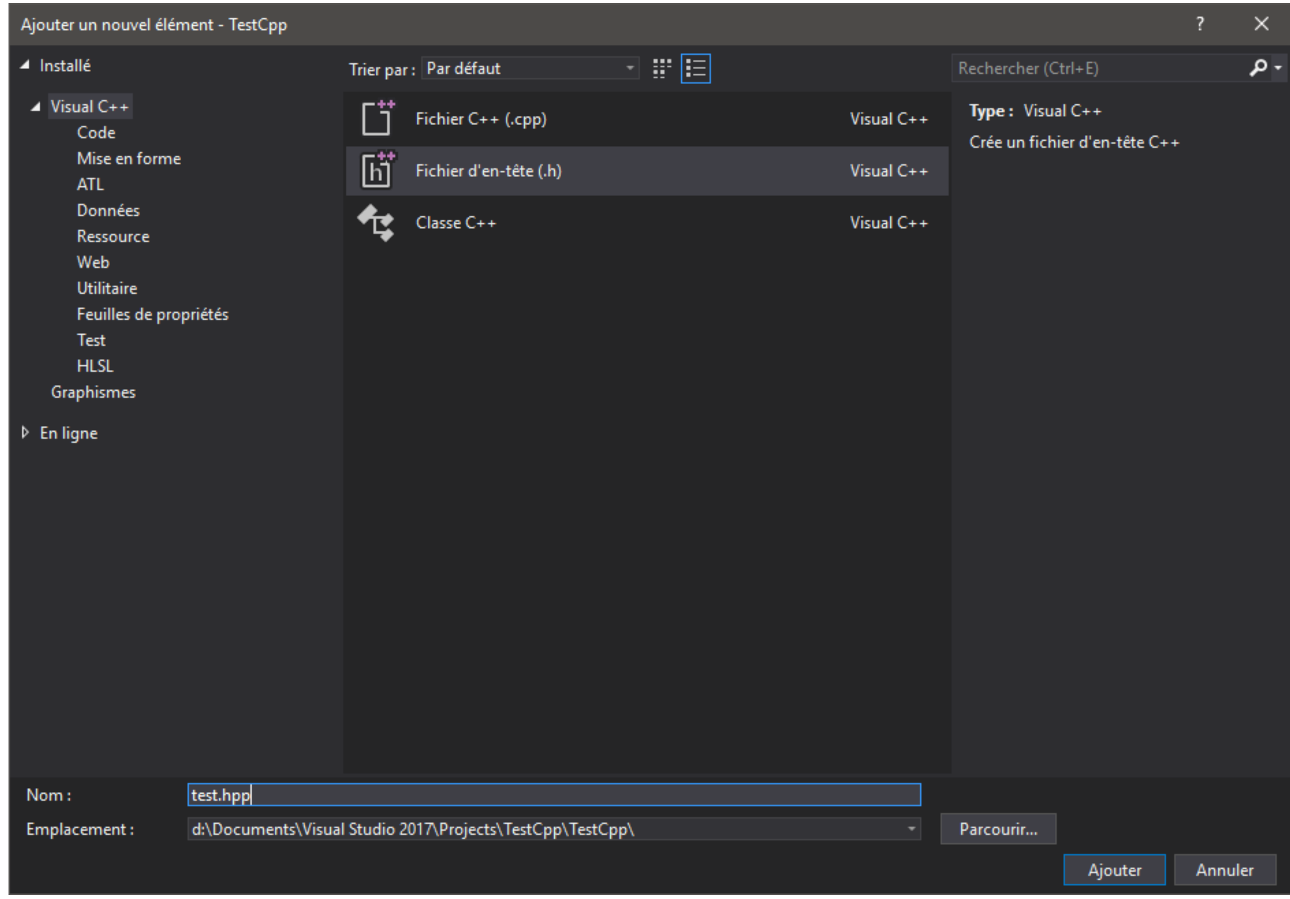


Figure 22.2. – Choisissez bien « Fichier d'en-tête (.h) ».

Choisissez bien Fichier d'en-tête (.h) puis donnez un nom à votre fichier. Ici, j'ai choisi `test.hpp`. Notez que j'ai changé l'extension, de `.h` vers `.hpp`, et je vous invite à faire de même. Enfin, cliquez sur Ajouter. Vous arriverez sur l'écran principal, avec le fichier ajouté au projet.

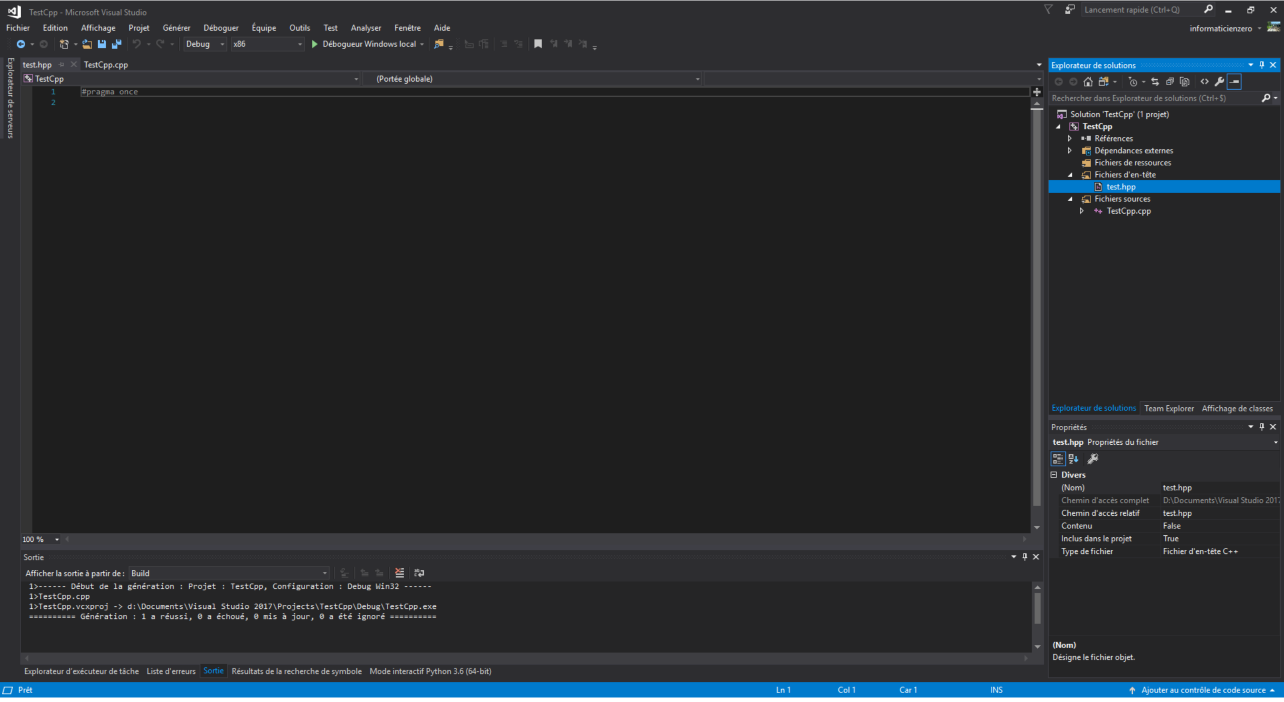


Figure 22.3. – Le fichier d'en-tête a bien été ajouté.

22.1.1.2. Sur QtCreator

Ouvrez votre projet avec Qt Creator, ce qui devrait vous donner quelque chose comme moi.

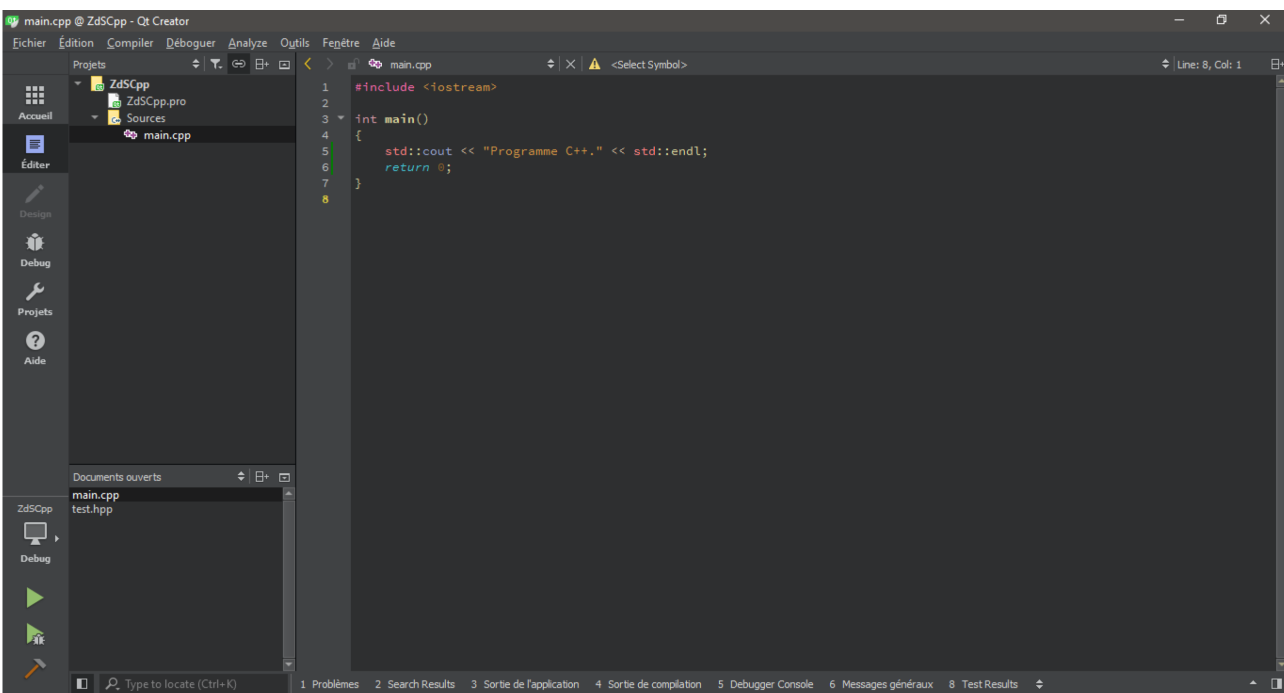


Figure 22.4. – Page principale de Qt Creator avant l'ajout d'un fichier d'en-tête au projet.

Faîtes un clic-droit sur le nom du projet (ici, *ZdSCpp*), puis cliquez sur Add new.... Vous tombez sur la fenêtre suivante.

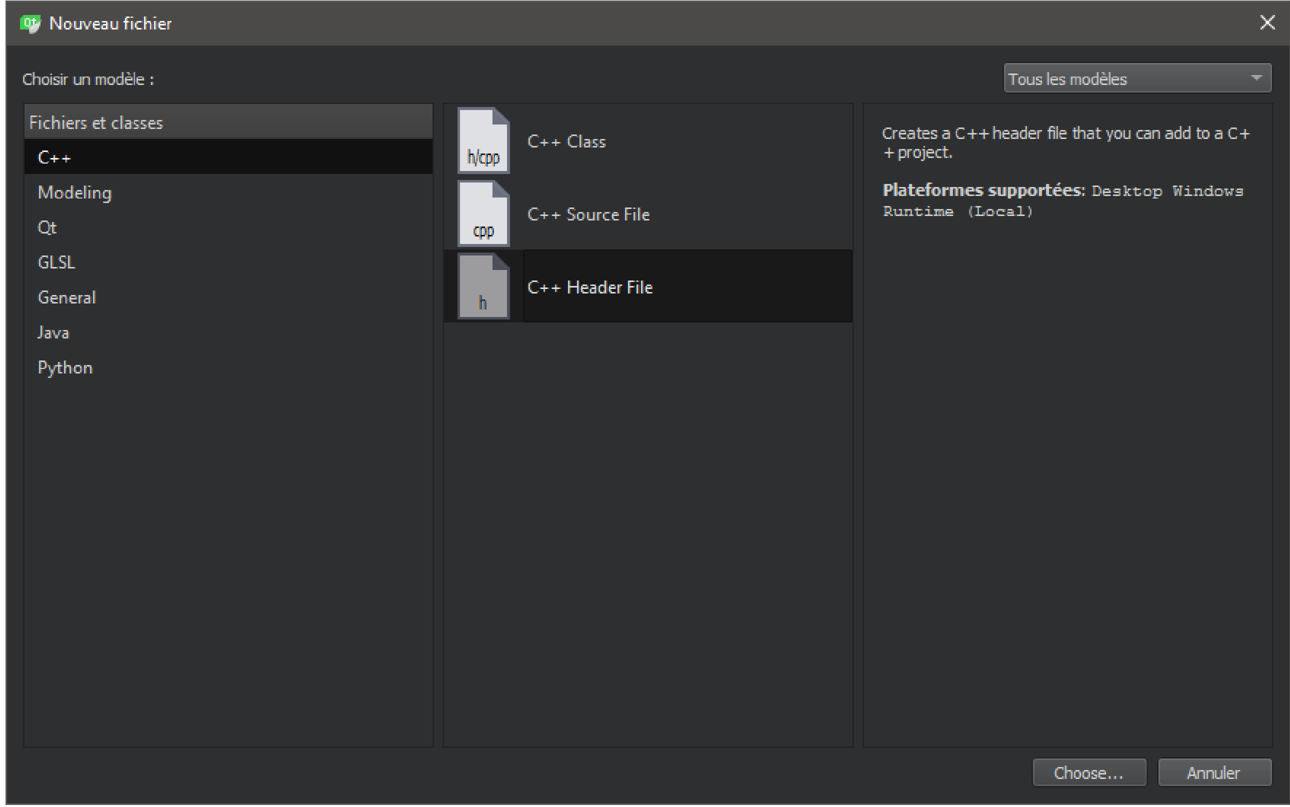


FIGURE 22.5. – Il faut choisir « C++ Header File ».

Cliquez sur C + Header File, puis sur Choose... en bas à droite. Vous passez au deuxième écran.

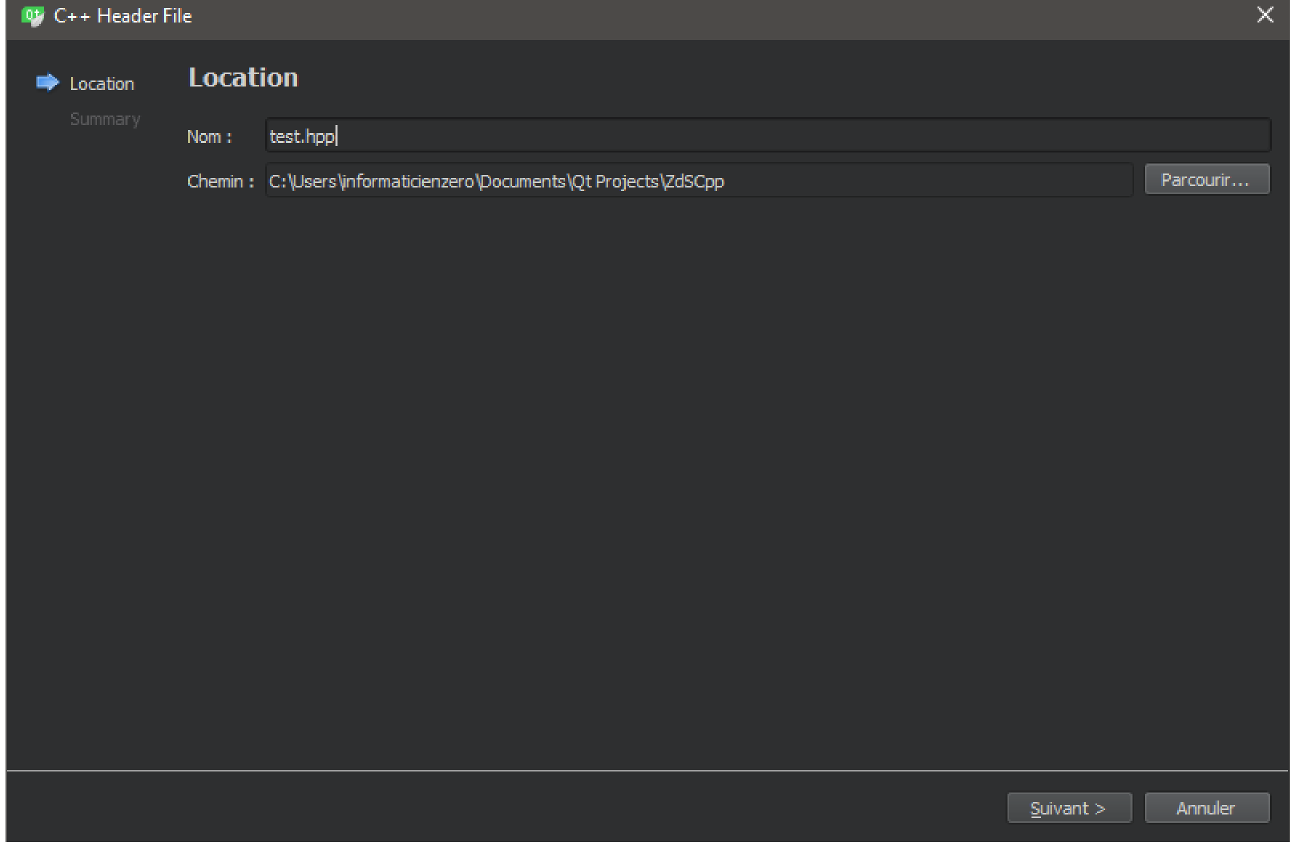


Figure 22.6. – Il faut maintenant choisir un nom.

Maintenant, donnez un nom à votre fichier. Ici, pour l'exemple, je l'ai appelé `test.hpp`. Cliquez ensuite sur Suivant et vous arriverez sur le troisième écran.

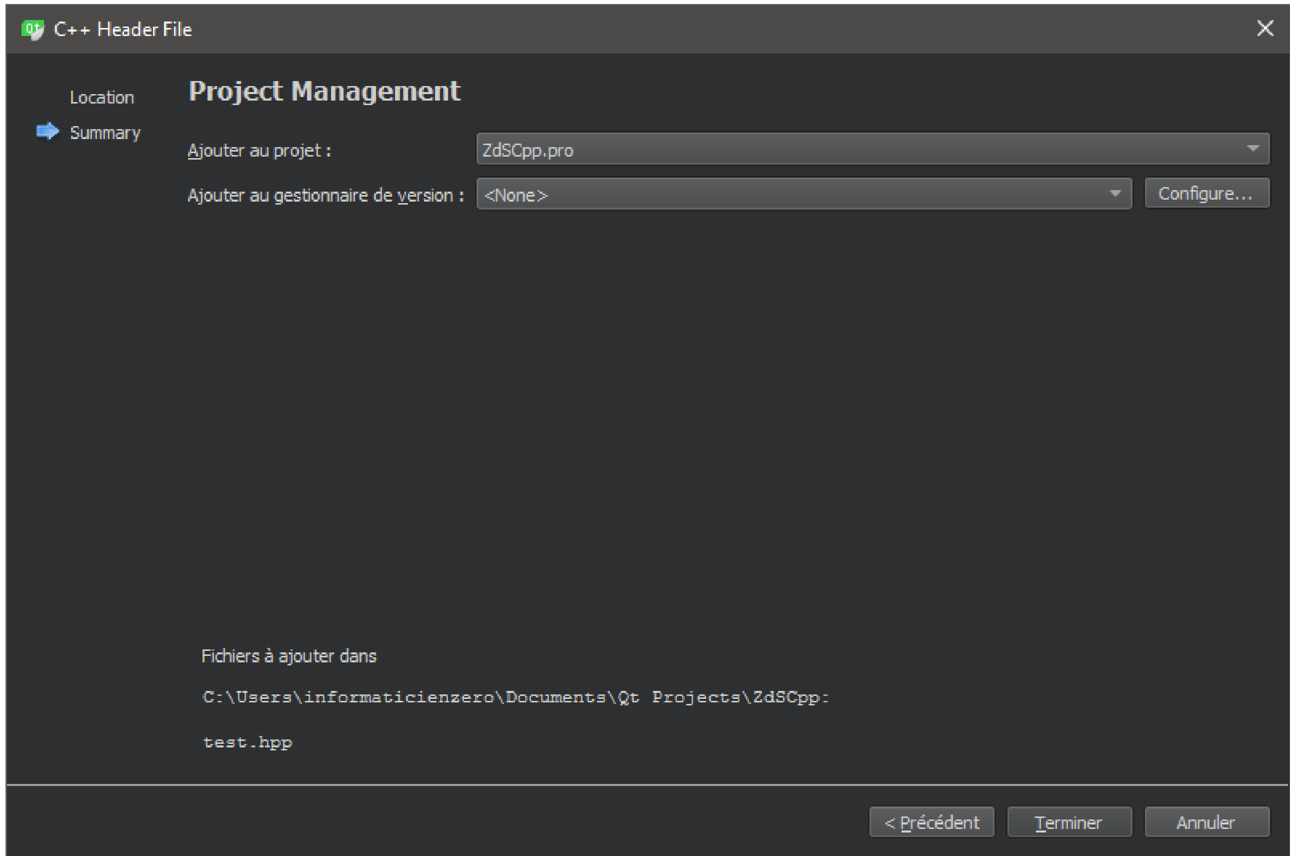


Figure 22.7. – Normalement, vous n'avez rien besoin de toucher.

Tout est bon, il ne reste plus qu'à cliquer sur Terminer et nous en avons fini avec l'ajout de fichiers d'en-tête. On retombe sur la page principale, avec notre fichier bien présent en haut à gauche, signe qu'il a bien été ajouté au projet.

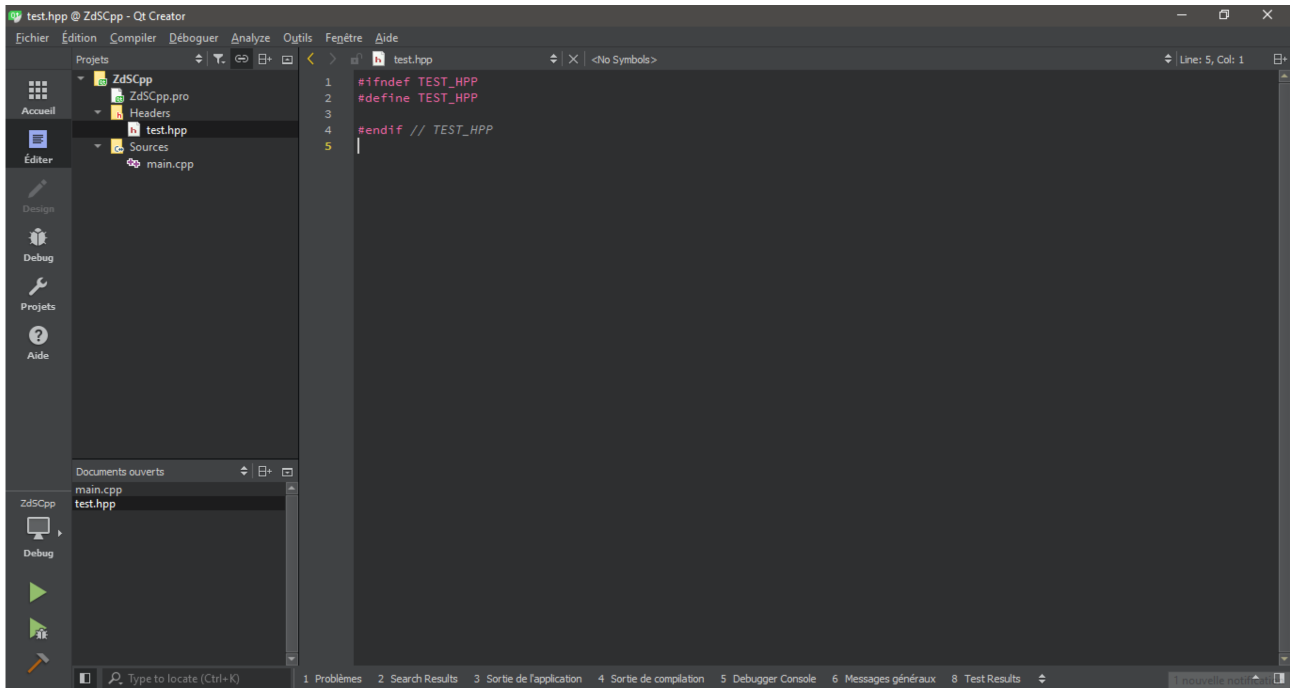


FIGURE 22.8. – Voilà, tout est bon.

22.1.1.3. En ligne de commande

Rien de compliqué, il suffit de créer un simple fichier `.hpp` et c'est tout.

22.1.2. Le fichier source

Le fichier source **va contenir les implémentations de nos fonctions**. La création d'un fichier source est exactement identique à celle de la création d'un fichier d'en-tête, à ceci prêt que, cette fois, c'est un fichier source.

Un fichier source peut inclure **autant de fichiers d'en-tête que nécessaire**. Il y a bien sûr celui qui contient les prototypes et les types qu'il définit, mais il peut en inclure d'autres.

22.1.3. Créons un lien

Il reste une étape importante : lier les fichiers d'en-têtes et sources. Pour cela, il suffit d'inclure le fichier d'en-tête concerné dans le fichier source concerné. Cependant, à la différence des fichiers de la bibliothèque standard qui s'incluent avec des chevrons, on utilise les guillemets doubles `"`. On doit aussi écrire le nom complet, **extension comprise**. Celle-ci est `.hpp`.

Ainsi, si vous nommez votre fichier `test.hpp`, vous l'inclurez dans le code en faisant comme suit.

Extension

Les fichiers d'en-tête peuvent aussi utiliser l'extension `.h`. C'est parfaitement valide. La différence n'est qu'une histoire de goût. Nous préférons `.hpp` à `.h` par analogie aux fichiers sources, qui se terminent par `.cpp` en C++ mais `.c` en C.

Voici un exemple tout bête mais fonctionnel de séparation en plusieurs fichiers. Pour le reproduire, créez les fichiers concernés dans votre projet et copiez-collez les codes ci-dessous.

test.hpp

```
```

test.cpp

```
```

main.cpp

```
```

Pour compiler ce programme avec Visual Studio ou Qt Creator, rien de bien compliqué, on lance la compilation et c'est tout. Pour ceux qui sont en console, il faut **ajouter chaque fichier .cpp** à la liste des fichiers à compiler. Quant aux **.hpp**, **il ne faut pas les inclure**, nous verrons pourquoi plus tard.

L'exemple suivant se compile ainsi.

```
```

22.2. La sécurité, c'est important

Avez-vous noté ces lignes nouvelles qui étaient automatiquement présentes lors de la création du fichier d'en-tête? Ce sont surtout celles de Qt Creator qui nous intéressent (celle de Visual Studio n'est pas du C++ standard). Je les réécris ci-dessous.

```
```

C'est qu'en fait, lorsque le compilateur voit une ligne `#include`, il ne réfléchit pas et inclut le fichier en question. Sauf que, si, par erreur, ou même volontairement, on inclut plusieurs fois le même fichier, on va se retrouver avec **plusieurs fois les mêmes déclarations de fonctions, de types, etc**. Autant dire que le compilateur ne va que très moyennement apprécier.

Afin de s'en prémunir, il faut écrire ces trois lignes, qui sont des directives de préprocesseur et ce dans chaque fichier d'en-tête que vous créerez. Vous pouvez remplacer `TEST_HPP` par ce que vous voulez, à condition que **ce nom soit unique**. Le plus simple est de le remplacer par le nom du fichier en question : s'il s'appelle `autre.hpp`, alors écrivez `AUTRE_HPP`. Le code de l'en-tête ira entre la deuxième et la troisième ligne.

Ainsi, le fichier d'en-tête de l'exemple de la section précédente n'est pas correct. Il faut le corriger en utilisant ce que nous venons d'apprendre.

22.3. Découpons le TP !

Pour s'exercer, on va prendre le corrigé du TP précédent et on va l'organiser en plusieurs fichiers. Commençons par le commencement, séparons prototypes et implémentations. On aura alors l'esprit plus clair pour réorganiser tout ça une fois qu'on aura uniquement les prototypes sous les yeux.

Séparation prototype / implémentation

On peut former des groupes de types et fonctions liées.

— Les fonctions utilitaires : pour l'instant, on n'a que `traitement_chaine`, mais elle pourrait être rejointe par d'autres lors de l'évolution du projet.
— Les types de base `Artiste`, `Album` et `Morceau` et les fonctions les manipulant.
— Les tests unitaires correspondants.
— Le type `Discographie` et ses fonctions de manipulation.
— Les types et fonctions concernant l'exécution des commandes.

On va donc créer cinq paires de fichiers pour réorganiser notre code en conséquence : `utils`, `donnees_disco`, `donnees_disco_tests`, `discographie`, `systeme_commandes`.

Solution

Comme d'habitude, il n'y a pas qu'une seule solution en programmation. L'organisation que je propose en est une, mais l'essentiel est juste d'avoir un code organisé de manière cohérente.

Cela donne donc les fichiers suivants.

Utilitaire

Données discographie

Tests unitaires

Discographie

Commandes

Notre fichier `main.cpp` devient alors très simple.

Et voilà ! On a un code organisé en plusieurs fichiers, prêt à évoluer !

22.4. Les avantages du découpage en fichiers

Le découpage en fichiers permet de mieux organiser le code, certes. Mais cela ne s'arrête pas là ; il y a d'autres avantages qu'on va voir maintenant.

22.4.1. Une structure de projet plus visible

Imaginez un projet encore plus conséquent que le T.P précédent. Tout est dans le même fichier. Compliquer après d'y voir un ordre quelconque. Si, par contre, le projet est découpé intelligemment en plusieurs fichiers, vous pourrez, d'un coup d'œil, **comprendre la structure du projet** et mieux orienter vos recherches.

Le projet Zeste de Savoir , par exemple, est découpé en plein de fichiers, qui séparent les tests, l'affichage, la manipulation de la base de données, etc. Un développeur ayant des connaissances dans les technologies utilisées sait donc, d'un rapide regard, où il doit orienter ses recherches.

22.4.2. Une meilleure abstraction

L'avantage de séparer l'implémentation des prototypes et des définitions de type, notamment avec les fichiers d'en-tête, c'est que **l'utilisateur ne dépend plus de détails techniques** mais de fonctions générales. Si besoin est, on peut changer du tout au tout la façon d'écrire nos fonctions, pour les rendre plus sûres ou plus rapides et l'utilisateur n'y verra rien.

Vous, apprentis développeurs C++, dépendez des prototypes et abstractions de la bibliothèque standard. Par contre, en coulisse, l'implémentation n'est pas la même selon que vous utilisez Visual Studio, GCC ou Clang. Mais ça n'est pas un problème. **Nous dépendons d'abstractions, non de détails**. C'est un des grands principes d'un programme bien conçu.

De plus, puisque les détails techniques sont cachés dans les fichiers sources, les fichiers d'en-tête sont bien plus lisibles et n'exposent que les fonctions et données publiques qui intéressent leur utilisateur.

22.4.3. Une meilleure modularité

Séparer en plusieurs fichiers, c'est aussi autoriser des portions de codes à être réutilisées dans d'autres endroits. Vous le savez très bien, puisque vous faites de nombreux `#include` d'en-têtes de la bibliothèque standard. Il serait plus difficile et bien plus lourd d'utiliser `std::string` ou `std::vector` s'il n'y avait pas les fichiers d'en-tête correspondants, n'est-ce pas ?

Rien que dans le T.P précédent, le fait d'avoir séparé la fonction de traitement des chaînes de caractères, pour la mettre dans un fichier à part, nous autorise à la réutiliser ailleurs que ce qu'on avait originellement prévu.

22.5. Le cas des templates

Il me reste un cas particulier à aborder, celui des *templates*. Dans le chapitre qui leur est consacré, nous avions dit que le compilateur va instancier une fonction *template* chaque fois qu'elle est appelée avec un type réel différent. Mais pour l'instancier, le compilateur a besoin de son **implémentation complète**. Du coup, un code comme celui-ci ne peut pas fonctionner.

conteneur.hpp

conteneur.cpp

main.cpp

Le compilateur va en effet se plaindre (avec une erreur cryptique, soit dit en passant), car il essaye d'instancier le *template* `afficher` mais ne trouve nulle part son implémentation. Afin de compiler, il faut que **toutes nos fonctions *templates* soient déclarées et implémentées directement dans les fichiers d'en-tête**. C'est contraire à ce que nous venons d'apprendre dans ce chapitre, mais c'est pourtant ce qu'il faut faire.

conteneur.hpp

?

Lecteur attentif

Il n'y a vraiment pas d'autres solutions ? Parce que tu as dis toi-même que c'était bien que les détails techniques restent cachés dans les fichiers sources.

Eh bien figurez-vous que si ! En fait, il y a une astuce toute simple, qui consiste à déclarer les fonctions dans le fichier d'en-tête, mais à **les implémenter dans un fichier séparé** (souvent avec l'extension `.tpp`), fichier qui sera **inclut à la fin du fichier d'en-tête**. Ainsi, on cache les détails techniques dans un autre fichier tout en laissant la compilation se faire.

conteneur_impl.tpp

conteneur.hpp

main.cpp

22.5.1. En résumé

- Découper en fichiers permet de séparer les définitions des implémentations.
- Les fichiers d'en-tête contiennent les définitions et portent, dans ce cours, l'extension `.hpp`.
- Les fichiers source contiennent les implémentations et les détails techniques. Ils terminent par l'extension `.cpp`.
- Lors de la création de fichiers d'en-tête, il faut bien penser à les sécuriser avec les directives de préprocesseur abordées.
- Découper son projet en fichiers permet une meilleure modularité, une meilleure réutilisation, une structure de projet plus visible.
- Dans le cas des *templates*, il faut que l'implémentation soit écrite directement dans le fichier d'en-tête, soit dans un fichier séparé qui lui-même sera inclut dans le fichier d'en-tête concerné.

Vous avez maintenant un certain niveau et êtes capables de faire beaucoup de choses. Bien entendu, votre apprentissage ne s'arrête pas là et je vous invite à continuer ce cours à nos côtés. Mais prenez quelques instants pour mesurer vos nouvelles compétences et apprécier les efforts que vous avez fournis.

Contenu masqué

Contenu masqué n°60 : Séparation prototype / implémentation

Retourner au texte.

Contenu masqué n°61 : Utilitaire

/utils.hpp

```
```

/utils.cpp

```
```

Retourner au texte.

Contenu masqué n°62 : Données discographie

/donnees_disco.cpp

```
```

/donnees_disco.cpp

```
```

Retourner au texte.

Contenu masqué n°63 : Tests unitaires

/donnees_disco_tests.hpp

```
```

/donnees_disco_tests.cpp

```
```

Retourner au texte.

Contenu masqué n°64 : Discographie

/discographie.hpp

```
```

/discographie.cpp

```
```

Retourner au texte.

Contenu masqué n°65 : Commandes

/systeme_commandes.hpp

```
```

/systeme_commandes.cpp

```
```

Retourner au texte.

Quatrième partie

Interlude - Être un développeur

L'art de développer ne se limite pas à la maîtrise du langage utilisé. Il y a, en effet, bien d'autres aspects dans le développement que la pure programmation et c'est ce que nous allons découvrir dans cette partie. Les outils que vous y découvrirez vous donneront une grande autonomie pour mener à bien des projets seuls. Vous serez alors à même de créer des programmes mine de rien assez complexes.

23. Avant-propos

Nous avons découvert de très nombreuses notions depuis le début de ce tutoriel, et avec elles nous avons touché du doigt certains aspects importants du développement informatique. Il est temps de revenir sur ce que nous avons fait et de l'analyser avec un peu plus de recul. Si nous insistons autant sur ces aspects, c'est parce qu'être un bon développeur nécessite plus que d'écrire du C++.

23.1. Ce que nous avons appris

Jusqu'à présent, nous avons abordé plusieurs aspects de la programmation en C++, que ce soit les notions de base comme les conditions, les boucles, ou encore des notions plus avancées comme l'utilisation d'outils fournis par la bibliothèque standard, ainsi que la création de fonctions et de structures de données personnalisées. Vous avez ainsi déjà acquis un arsenal de programmation assez complet pour faire des programmes relativement complexes. Le T.P de la partie précédente est là pour le prouver.

Cependant, nous avons vu dans ces deux parties bien plus que de simples notions de C++. En effet, nous avons touché du doigt **certains aspects essentiels du développement**. Plutôt que de simplement apprendre à programmer en C++, nous avons découvert une certaine manière de réfléchir qui traduit les qualités d'un bon développeur : le désir d'avoir toujours un code de qualité, à travers **un travail de conception poussé** et **des bonnes pratiques**. Citons, parmi celles-ci, les **tests unitaires** pour vérifier que notre code fait ce qu'on lui demande, les **contrats** pour définir clairement ce que les différents intervenants attendent comme entrées et sorties et les **exceptions** pour remonter des problèmes à l'utilisateur.

Ce sont ces qualités qui font qu'un développeur sera non seulement capable d'utiliser son langage, mais surtout **de mener ses projets à bien**. Pour être un bon marin, il ne suffit pas de savoir utiliser la voile et le gouvernail, encore faut-il savoir naviguer, s'orienter dans la mer. De même, il ne suffit pas pour un développeur de maîtriser ses outils pour mener sa barque à bon port.

Je pense que vous avez commencé à vous rendre compte de ces aspects-là, notamment grâce à la partie II qui insiste fortement sur l'amélioration de la qualité du code. Et c'est l'une des choses les plus importantes à retenir des deux premières parties. Dans cet interlude, nous allons justement découvrir de nouvelles notions qui s'inscrivent dans cette optique, à savoir faire de vous **de bons développeurs** et pas simplement de bons programmeurs.

23.2. Différents paradigmes

Nous avons souvent dit que tout problème de développement a de nombreuses solutions, l'essentiel étant que les choix que l'on fait soient guidés par la volonté de produire un code propre et robuste à travers une bonne conception.

Eh bien cette diversité fait que l'on classe les manières de programmer en différentes catégories, appelées paradigmes. Le C++ étant un langage multi-paradigmes permettant de programmer de manière très libre, il nous laisse choisir parmi plusieurs de ces manières de programmer. Nous en avons déjà croisé quelques-unes.

23.2.1. Le paradigme impératif

Le paradigme impératif est le principal paradigme que l'on a utilisé. C'est aussi celui de beaucoup d'autres langages de programmation, dont notamment les premiers inventés. Dans cette approche, le programme est une suite d'instructions qui s'exécutent linéairement et le font avancer. Ce programme inclut les notions de conditions, boucles, fonctions, etc. Une grande partie des notions que l'on a abordées s'inscrit, au moins partiellement, dans ce paradigme.

Voici un exemple d'un code écrit typiquement avec le paradigme impératif, qui met chaque élément d'un tableau au carré.

23.2.2. Le paradigme fonctionnel

Le paradigme fonctionnel est aussi en partie couvert par le langage C++. C'est un paradigme dans lequel, comme son nom l'indique, **les fonctions jouent un rôle prépondérant**. D'ailleurs, dans les langages dit « fonctionnellement purs », la modification de variable peut ne pas exister et tout est réalisé avec des calculs sur résultats de fonctions ! Nous avons principalement vu ce paradigme à travers les lambdas, qui sont un concept venant tout droit des langages fonctionnels. De manière plus générale, le fait de pouvoir manipuler les fonctions comme des variables vient du paradigme fonctionnel.

Le code suivant est une réécriture de l'exemple précédent, dans un style plus fonctionnel.

23.2.3. Le paradigme générique

Le paradigme générique consiste principalement en l'idée que parfois, on se fiche du type réel des données que l'on manipule. Ce qui compte, c'est la manière dont elles peuvent être utilisées. Nous l'avons principalement appréhendé sous deux formes.

- Les algorithmes standards qui peuvent s'appliquer à tout un tas de collections différentes.
- Les fonctions *templates* qui permettent d'écrire du code fonctionnant avec plusieurs types, du moment qu'on peut les manipuler de cette manière.

En reprenant les exemples précédents, utilisons le paradigme générique. Voici un résultat qu'on peut obtenir.

23.2.4. La programmation par contrat

Bien que ce ne soit pas aussi poussé que dans d'autres langages comme Eiffel ou le D, en C++, il est possible de faire de la programmation par contrat, comme nous l'avons vu dans le chapitre sur la gestion des erreurs. Comme nous avons pu le constater, les principes de la programmation par contrat se combinent très bien avec les autres paradigmes, donc n'hésitez pas à en user et abuser.

C++20 et PpC

La prochaine version de C++ (C++20) devrait inclure de nouveaux outils pour faire de la programmation par contrat en C++. Cela permettra de l'utiliser de manière encore plus poussée.

Maintenant que l'on a plus de recul, on voit que le C++, contrairement à d'autres langages plus restrictifs, met à notre disposition tout un panel d'outils très variés pour écrire du code de qualité. Nous pouvons prendre un peu de chaque paradigme et se servir de ses points forts. Dans la suite du cours, nous introduirons un nouveau paradigme, lui aussi très important en C++ et qu'on appelle la **programmation orientée objet**.

Remarque au passage

Ce n'est pas une critique envers d'autres langages que de dire qu'ils sont moins permissifs. C'est parfois un avantage d'utiliser un langage qui fait des choix forts. Simplement, l'un des attraits du C++ est cette permissivité.
Mais c'est aussi l'un de ses plus grands dangers. Comme on peut faire beaucoup de choses, on peut très facilement faire n'importe quoi ! C'est pourquoi il faut être particulièrement

concentré sur le fait de produire de code propre et robuste. D'où le fait que nous insistions régulièrement dessus.

23.3. Brancher le cerveau avant tout

Nous l'avons compris, le C++ nous offre de nombreux outils, ainsi que la liberté d'utiliser et mélanger plusieurs paradigmes. Mais tout cela n'est qu'outils et instruments. Il faut les assembler intelligemment et harmonieusement pour produire de bons programmes. Pour ça, il faut, avant de se ruer sur le clavier et de martyriser énergiquement ses pauvres touches, **réfléchir et concevoir**.

On prend habituellement un stylo et un papier[2], puis **on se pose des questions** sur le programme sur lequel on travaille. On réfléchit aux fonctionnalités du programme, ce qu'il rendra comme service, ce qu'il ne fera pas, ce qu'il est capable de gérer, etc. On se demande quels sont les cas d'erreurs, les entrées invalides, ce qui peut potentiellement poser problème, etc.

Les contrats

Nous avons déjà commencé à faire ça, notamment avec les contrats et la réflexion sur nos entrées et sorties.

On découpe aussi de grosses fonctionnalités **en plus petites** afin qu'elles soient plus facilement abordables. Un logiciel de traitement d'images peut ainsi être découpé en sous-tâches comme l'ouverture d'un fichier PNG, la modification d'un unique pixel, la sauvegarde du PNG modifié, etc.

C'est là que le fait de noter ses idées se révèle utile. On peut comparer plusieurs solutions à un même problème, on peut visualiser plus facilement à quoi ressemblera le programme final, on peut écrire du pseudo-code pour s'aider à mieux comprendre un algorithme complexe, etc. Et puis, personne n'ayant une mémoire infaillible, noter ses idées aide à s'en rappeler.

Cependant, concevoir, anticiper, tout cela ne signifie pas que le programme est figé et qu'il n'évoluera pas d'un pouce. Au contraire, réfléchir avant de coder permet de mieux prévoir des aspects qui pourront être amenés à changer. Cela rend le programme **plus évolutif**.

Planification ou souplesse ?

C'est un exercice délicat que de trouver l'équilibre entre le néant et trop de spécifications, entre une trop grande rigidité et une absence de réflexion et de conception. Ne vous inquiétez pas. Comme beaucoup de choses, tout vient en pratiquant. L'étude de cas vous donnera un exemple concret.

2. Enfin, un document numérique ça marche aussi. Question de goût.

23.4. Savoir se débrouiller

Être un bon développeur, c'est être **autonome** et **se débrouiller**. Cela ne veut pas dire que les développeurs ne se parlent jamais entre eux, vivant reclus les uns des autres. Au contraire, l'existence même de ce tutoriel et de la plateforme Zeste de Savoir sont des preuves que l'échange est important.

En fait, ce qu'il faut comprendre, c'est qu'**il ne faut pas s'attendre à ce que tout arrive cuit dans le bec**. Un bon développeur ne compte pas sur les autres pour faire son travail. Il n'attend pas de réponse toute faite. Quand il est bloqué, il réfléchit d'abord et demande ensuite.

Si vous êtes vraiment bloqués sur un problème, bien entendu que vous pouvez demander un coup de pouce, notamment sur Zeste de Savoir. Mais quand cela arrivera, montrez ce que vous avez tenté, les erreurs qui vous bloquent, si vous avez déjà fait des recherches dessus, etc. Et vous verrez, vous aurez de l'aide et des réponses.

À venir

Dans cette partie, nous verrons notamment une technique appelée le **débugage**, qui permet de voir pas-à-pas l'exécution du code pour trouver là où une erreur se cache. Avec cette compétence, vous pourrez déjà bien avancer seuls dans la recherche de la solution.

En conclusion, laissez-moi vous lister quelques qualités, en plus de l'autonomie, qui feront de vous de meilleurs développeurs.

— **Patience**. Il en faut pour apprendre, pour comprendre, pour faire et corriger des erreurs, pour chercher des solutions, pour ne pas s'énerver quand ça ne marche pas.
— **Persévérance** et **motivation**. Développer, c'est parfois rencontrer des problèmes qui nous dépassent, qui semblent trop complexes à première vue, ou bien des notions trop abstraites. Ces qualités aideront à ne pas abandonner à la première difficulté.
— **Humilité**. Un bon développeur accepte de ne pas avoir réponse à tout et donc de demander de l'aide quand il n'y arrive vraiment pas. Il accepte les conseils et les corrections qu'on lui donne. Il accepte de ne pas se reposer sur ses acquis, se remet régulièrement à jour et en question. Cela lui évite les réponses méprisantes, arrogantes et hautaines qu'on voit parfois sur certains forums.

23.4.1. En résumé

— Nous avons découvert plein de choses sur le langage C++ pendant ce cours, mais on a aussi mis en avant des aspects plus généraux du développement, comme l'écriture de tests unitaires, de contrats ou la réflexion à mener sur les entrées / sorties.
— Le C++ est un langage très riche qui nous permet de programmer de manières très différentes.
— Cette liberté est une force mais aussi un danger, c'est pourquoi une bonne conception et du code de qualité sont primordiaux.

— Être un bon développeur demande aussi plusieurs qualités, dont la persévérance, l'autonomie et la patience.
— La suite de cette partie nous aidera à tendre vers cet idéal en nous apprenant de nouvelles notions pour mieux développer.

24. Mais où est la doc ?

Cette chère documentation est une des plus vieilles amies du développeur. Parfois très complète, parfois vraiment courte, parfois pas présente du tout (pas de bol), elle est là pour **le guider**, **répondre à ses questions** et lui apprendre comment utiliser tel ou tel type, ce qu'attend telle ou telle fonction, etc. Heureusement, dans notre cas, il existe une excellente documentation pour C++, en anglais certes (sachez que la version française est une traduction automatique), mais simple à comprendre. Et elle n'aura bientôt plus de secret pour vous.

Nous allons apprendre à **lire la documentation C++** ainsi qu'à **écrire celle de nos programmes**.

24.1. Lire une page de doc

24.1.1. À l'arrivée

Vous êtes donc sur la documentation et tant de liens vous laissent perplexes. Comment on s'y retrouve là-dedans ? Par où chercher l'information ? Le site étant classé en plusieurs parties, tout dépend de ce que vous cherchez. Certaines vont plus nous intéresser, dans le cadre de ce cours, que d'autres.

— ***Headers*** / **Fichiers d'en-tête standard**. Cette page recense tous les fichiers d'en-tête de la bibliothèque standard. Nous en reconnaissons plusieurs, comme `<iostream>`, `<string>` ou `<vector>`. Ils sont classés par catégorie, en fonction de s'ils concernent la manipulation de chaînes de caractères, les conteneurs, etc.
— ***String library*** / **Chaînes de caractères**. Ici, nous allons retrouver tout ce qui concerne le célèbre `std::string`.
— ***Containers library*** / **Conteneurs**. Ici, vous avez un lien direct vers tous les conteneurs qui existent. Vous reconnaitrez notamment `vector`, `array` et `list`.
— ***Algorithms library*** / **Algorithmes**. Vous en connaissez certains, mais il en reste d'autres, tous listés ici.

Je vous cite ici les autres catégories, pour information et pour satisfaire votre curiosité débordante.

— ***Utilities library***. Correspond aux utilitaires pour gérer les types, la mémoire, les erreurs, les dates et temps, etc.
— ***Iterators library*** / **Itérateurs**. Vous connaissez déjà les itérateurs. Ici, on retrouve la documentation de tout ceux qu'on connait, plus bien d'autres encore.
— ***Numerics library*** / **Numérique**. C'est tout ce qui touche à la manipulation des nombres (complexes, aléatoires, etc).

— ***Input/output library*** / **Entrées/sorties**. On a déjà un peu manipulé, mais il reste encore beaucoup de choses.
— ***Localizations library*** / **Internationalisation**. Nous n'en parlerons pas, mais il y a ici des éléments permettant d'adapter son programme à d'autres langues et cultures.
— ***Filesystem library*** / **Système de fichiers**. Pour permettre la manipulation de dossiers et de fichiers de manière standard.
— D'autres encore, d'un niveau trop avancé pour être expliqués maintenant.

Au fur et à mesure que vous progresserez en C++ et gagnerez en autonomie, vous pourrez vous débrouiller seuls dans la documentation, car celle-ci est organisée de manière logique.

24.1.2. `vector` – Retour sur le plus célèbre des conteneurs

25. Présentation

Commençons en examinant la page dédiée à `vector`. Il y a une très grande quantité d'informations, mais toutes ne nous sont pas utiles. Commençons par lire à quoi ressemble un `vector`.

Vous reconnaissez la syntaxe des templates. Seulement, vous aviez l'habitude jusque là de déclarer des objets de type tableau de la manière suivante `std::vector<int> tableau`, alors qu'ici vous voyez qu'il y a deux types génériques attendus. C'est parce que **le deuxième est renseigné avec une valeur par défaut**.

— `T` est le type des éléments.
— `Allocator` est utilisé pour l'acquisition de la mémoire pour stocker les éléments.

On ne modifie le deuxième qu'en cas de besoins très particuliers, notamment quand on veut manipuler la mémoire d'une manière précise, donc pour nous, ça ne présente aucun intérêt. On laisse donc la valeur par défaut et on retrouve ainsi la syntaxe habituelle.

25.0.0.1. Les fonctions membres

Sautons plusieurs détails et passons directement à la section ***Member functions*** / **Fonctions membres**. Ici sont listées toutes les fonctions que l'on peut appliquer à un `vector` avec la syntaxe `mon_vecteur.fonction()`. Certaines vous sont familières, comme `push_back` ou `clear`. Si vous cliquez sur l'une d'elles, vous aurez tous les détails la concernant. Examinons ces deux exemples.

Chaque page de fonction présente d'abord tous les prototypes de la fonction. Comme vous le savez déjà, avec les surcharges, il peut exister plusieurs fonctions avec le même identificateur mais des paramètres différents. C'est le cas ici pour `push_back`. Si une fonction est apparue en même temps qu'une nouvelle norme de C++, ou si elle a été supprimée, cela est précisé à la fin de la ligne.

Dans le cas de la fonction `push_back`, nous avons deux surcharges, dont la deuxième n'est disponible qu'à partir de C++11.

Petite explication

La syntaxe du deuxième prototype doit vous surprendre. En effet, nous ne l'avons pas encore abordée. Nous le ferons plus loin dans le cours, mais sachez que, entre autres, la deuxième version est appelée **quand vous lui donnez directement un littéral** (`push_back(5);`), alors que si vous passez une variable, ça sera le premier prototype (`int const a { 5 }; push_back(a);`). Pour faire simple, cela permet d'optimiser le code lorsqu'il est appelé avec une valeur temporaire.

25.0.0.2. Les fonctions non-membres

Sur la page de `std::vector`, on trouve d'autres fonctions, dites **non-membres**. Celles-ci utilisent la syntaxe `fonction(mon_vecteur)`, en opposition à ce que nous avons vu au sous-titre précédent mais en relation avec des fonctions que vous connaissez, comme `std::size` ou `std::begin`. C'est ici que vous trouverez la plupart des opérateurs surchargés, notamment.

On trouve, par exemple, `std::swap`, qui permet d'échanger le contenu de deux vecteurs, ou bien des opérateurs de comparaison.

25.0.0.3. Les paramètres

Revenons sur la page `push_back`. La sous-section **Parameters** y décrit les différents paramètres qu'une fonction attend. Rien de bien surprenant pour vous dans le cas de `push_back`, il s'agit simplement de la valeur à insérer. Ensuite, la sous-secton **Return value** décrit le résultat, qui, dans notre cas, n'existe pas puisque la fonction renvoie un `void`.

25.0.0.4. Les exceptions

Intéressons-nous maintenant à la section **Exceptions**. Comme vous le savez, C++ permet d'utiliser les exceptions et certaines fonctions de la bibliothèque standard le font. Cette section décrit le comportement attendu au cas où une fonction viendrait à lancer une exception. En effet, toutes ne réagissent pas de la même manière.

- Certaines ont une **garantie forte** (*strong exception guarantee*). Elles garantissent que, si une exception est levée, alors rien n'est modifié et tout reste comme avant le lancement de l'exception. C'est le cas de `push_back` : si une exception est levée, alors le tableau reste inchangé.
- D'autres ont une **garantie basique** (*basic exception guarantee*). Dans ce cas, la seule chose dont on est sûr, c'est qu'il n'y a pas de fuite de mémoire. Par contre, l'objet peut avoir été modifié.
- D'autres encore ont une **garantie pas d'exception** (*no-throw guarantee*). Aucune exception ne sera levée. Ces fonctions possèdent le mot-clef `noexcept` dans leur prototype. C'est le cas de la fonction `empty` par exemple.
- Enfin, il en existe qui n'ont **aucune garantie** (*No exception safety*). Tout peut se passer potentiellement. Ce genre de fonction est celle qui offre le moins de garantie et de protection.

Dans le cas où la fonction est `noexcept`, la section **Exceptions** peut être absente de la page de documentation. Cela dépend des cas.

Attention avec `noexcept`

Si une fonction spécifiée `noexcept` lève quand même une exception, **on a affaire à un UB**. En effet, ce mot-clef n'est nullement contraignant et n'empêche pas une exception d'être lancée.

25.0.0.5. Exemple(s)

Enfin, la section ***Example*** / **Exemple** donne un cas d'utilisation concret de la fonction exposée. On peut même modifier et exécuter le code en cliquant sur Run this code.

25.0.1. Les algorithmes

Continuons notre apprentissage en nous rendant sur la page des algorithmes. Nous en connaissons déjà quelques-uns, ce qui va nous faciliter la compréhension de la documentation. Notez que les algorithmes sont eux-mêmes classés en sous-catégories, comme « ***Non-modifying sequence operations*** / Non modifiants », « ***Modifying sequence operations*** / Modifiants », « ***Sorting operations*** / Tri », etc. Au début de chaque sous-catégorie se trouve le fichier d'en-tête à inclure pour utiliser les algorithmes sous-cités.

Prenons l'exemple de `std::sort`, que vous connaissez déjà. Si vous le cherchez dans la page, vous le trouverez dans la catégorie « *Sorting operations* », puisqu'il s'agit d'un algorithme de tri. On note en haut que le fichier d'en-tête requis est `<algorithm>`. Maintenant que nous avons ces informations, nous pouvons nous rendre sur sa page.

Une fois là-bas, on remarque qu'il existe quatre signatures différentes, chacune étant ordonnée avec un numéro. La première est celle que nous connaissons le mieux, puisque nous l'avons déjà vue. On remarque que le prototype change en fonction de si l'on compile en C++20 ou antérieur. La différence réside dans le mot-clef `constexpr`, que je détaillerai plus tard.

```

```

Le troisième lui ressemble, à ceci près qu'on a une possibilité de personnaliser l'algorithme de comparaison. Là encore, la seule différence entre pré-C++20 et post-C++20, c'est `constexpr`.

```

```

Les deux autres, à savoir le (2) et le (4), ont été introduits avec C++17 et permettent de traiter plusieurs instructions en parallèle plutôt que les unes à la suite des autres. C'est encore compliqué à notre niveau, donc je ne m'embêterai pas à les expliquer. De toute façon, à la fin de ce chapitre, vous aurez l'autonomie et les ressources nécessaires pour aller lire et pratiquer par vous-mêmes.

La section ***Parameters*** nous donne des indications supplémentaires sur les arguments attendus par `std::sort`. On y trouve notamment la forme que doit prendre la fonction transmise en paramètre de `comp`. Dans ce cas-ci, vous le saviez déjà, mais c'est dans cette section que vous trouverez la même info pour d'autres algorithmes.

Enfin, avant d'avoir un exemple en code, on nous informe que la fonction ne renvoie rien et qu'elle peut éventuellement lever une exception de type `std::bad_alloc`, nous indiquant par-là de quoi nous protéger.

25.0.2. Les chaînes de caractère au grand complet

Voici la page de documentation de `std::string` . Comme déjà dit lors de l'introduction de `using`, le type `std::string` est un alias d'un type `basic_string`, instancié pour les `char`. Ne vous étonnez donc pas en voyant la documentation.

25.0.2.1. Exemple de fonction membre

Vous pouvez voir, dans la section ***Member functions***, tout un tas de fonctions que nous ne connaissions pas encore. Certaines sont familières, comme `push_back` ou `find`. D'autres nouvelles, comme `insert` , que nous allons étudier.

En regardant les différents prototypes proposés, on distingue pas moins de 11 signatures différentes. On peut insérer plusieurs fois un même caractère (1), on peut insérer une chaîne entière (2)(4), des morceaux de chaînes grâce aux itérateurs (8) ou à des index (5), etc. Chacune est détaillée plus bas, avec explication de chaque paramètre attendu.

Toutes ces fonctions peuvent lever des exceptions. La documentation cite `std::out_of_range` si on transmet un index plus grand que la taille de la chaîne (logique) et `std::length_error` si on tente de créer une trop grande chaîne. Dans tous les cas, on a une garantie forte.

25.0.2.2. Exemple de fonction non-membre

Comme pour `std::vector`, on a ici des opérateurs de , mais d'autres encore. Ainsi, quand nous concaténons deux chaînes en faisant `"A"s + "B"s`, nous faisons appel à `operator+` .

25.0.2.3. Exemple d'autres fonctions associées

Tout ce qui permet de convertir une chaîne de caractères en entiers ou réels (et vice-versa) est listé sur cette page. Ces fonctions ne font pas partie du type `std::string`, mais sont tellement liées qu'elles sont réunies au sein du même fichier d'en-tête. On retrouve ainsi `stoi` , `stod` et `to_string` .

25.1. Exercices

Pour que vous gagniez en autonomie et que vous soyez capables de vous débrouiller seuls dans la documentation, il faut que vous vous entraîniez. Le but de cette section d'exercices est donc que vous appreniez à chercher une information dans la documentation. C'est important, car vous n'aurez pas toujours un tutoriel pour vous expliquer quoi et où chercher.

25.1.1. Remplacer une chaîne de caractère par une autre

J'aimerais pouvoir remplacer un morceau d'une chaîne de caractères par un autre. Disons que je veux remplacer une portion commençant à un index précis et d'une longueur précises. Le code suivant devra être complété pour pouvoir s'exécuter.

Correction

25.1.2. Norme d'un vecteur

Pour ceux qui ont quelques connaissances en vecteurs, vous savez comment en calculer la norme. Pour les autres, sachez que la norme d'un vecteur est « sa longueur » que l'on calcule simplement à l'aide de la formule suivante. Il me semble qu'il y a une fonction pour ça en C++, mais laquelle ?

Norme d'un vecteur

Si $\vec{v}$ est un vecteur de coordonnées $(x; y)$, alors sa norme $\|\vec{v}\|$ vaut $\sqrt{x^2 + y^2}$.

Indice

Correction

25.1.3. Nombres complexes

Ah, les nombres complexes! Saviez-vous que C++ permettait d'en utiliser? Eh oui, je peux calculer la racine carrée de -1 si ça me chante. En fait non, c'est plutôt vous qui allez le faire. Que diriez-vous de faire ce calcul puis d'afficher la partie réelle puis la partie imaginaire du résultat?

Indice

Correction

25.1.4. Transformations

Vous connaissez déjà l'algorithme `std::for_each`, qui applique une opération sur chaque élément d'un conteneur. Mais imaginez que nous ne voulons pas modifier le tableau lui-même mais une copie. Est-ce possible? À vous de trouver l'algorithme adapté.

Correction

25.2. Documenter son code avec Doxygen

Vous commencez maintenant à vous débrouiller avec la documentation officielle de C++. C'est une très bonne chose, car savoir lire et comprendre une documentation est un gros atout en informatique, vu que la très grande majorité des langages et des programmes viennent avec la leur. Mais les vôtres alors? C'est un peu vide n'est-ce-pas? La suite logique serait donc **d'apprendre à écrire la documentation de notre code**.

25.2.1. Installation des outils

Cela va se faire en utilisant un outil appelé Doxygen. Pour l'installer sur Windows, rendez-vous sur la page des téléchargements et prenez la dernière version disponible. Pour les utilisateurs de GNU/Linux, regarder dans vos dépôts. Le paquet devrait s'appeler `doxygen`, accompagné ou non de `doxygen-doxywizard`[3]. Une fois ceci fait, nous allons utiliser le **Doxygen Wizard**, l'interface graphique pour lancer la génération de la documentation.

3. Référez-vous à la documentation de votre distribution favorite pour savoir exactement quoi installer.

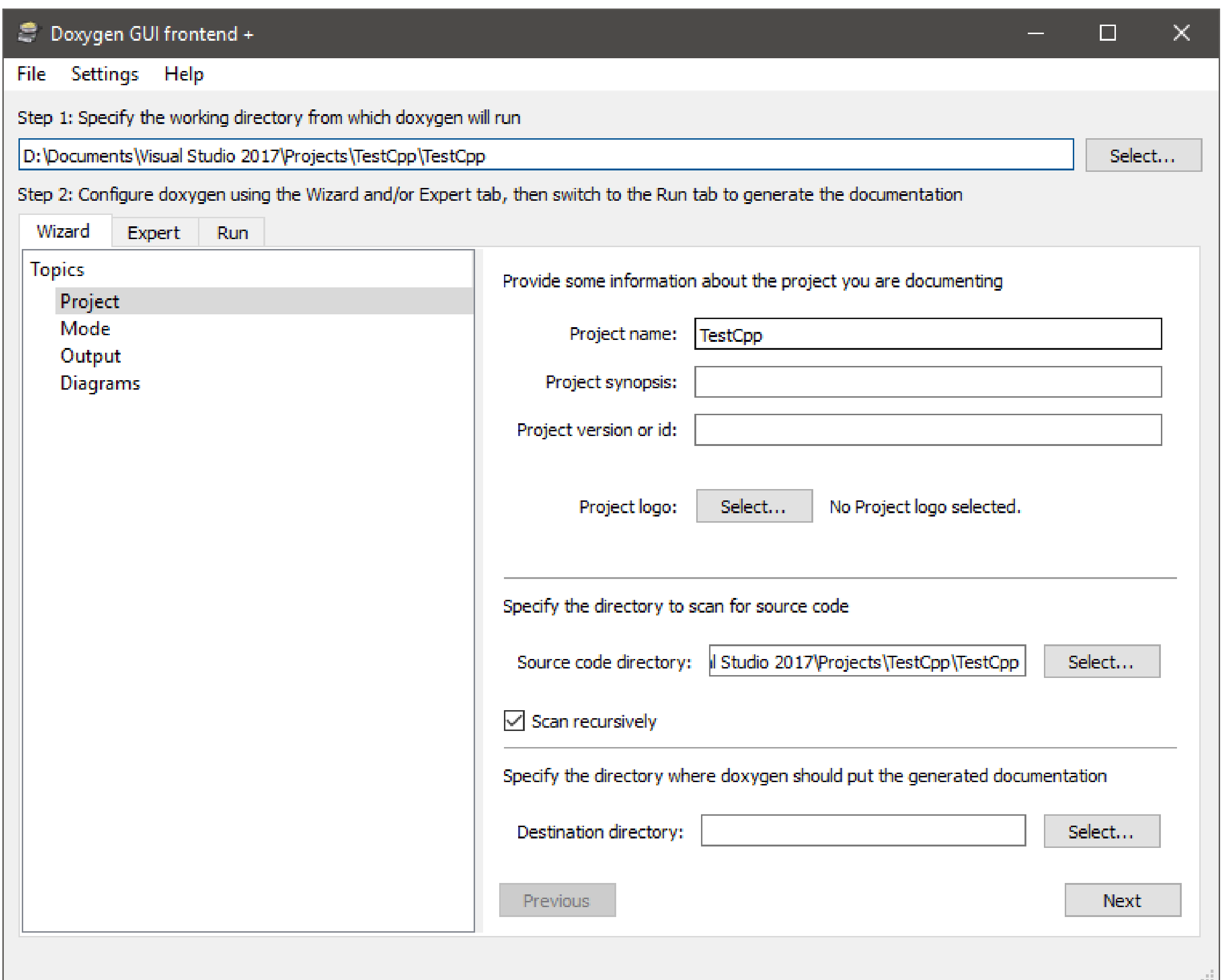


FIGURE 25.1. – Voici le « magicien Doxygen », si l'on traduit son nom en français.

La première étape est de **définir le chemin de sortie**, là où la documentation sera écrite. Personnellement, je la mets dans le même dossier que le code, mais vous êtes libres de faire comme vous le souhaitez.

Ensuite, on s'assure qu'on est bien sur l'onglet « *Wizard* » où l'on va pouvoir configurer plusieurs options. Les trois premiers champs sont facultatifs, mais permettent de définir le nom du projet, son but et son numéro de version, dans le cas où il y aura plusieurs versions du programme et donc de la documentation disponibles. Vous pouvez vous amuser à mettre un logo aussi.

Il faut ensuite préciser **où se situe le code source exactement**. Dans mon cas, il est dans le même dossier qu'indiqué plus haut, mais dans d'autres, il peut être rangé dans un sous-dossier. Mettez le chemin du dossier qui contient votre code source.

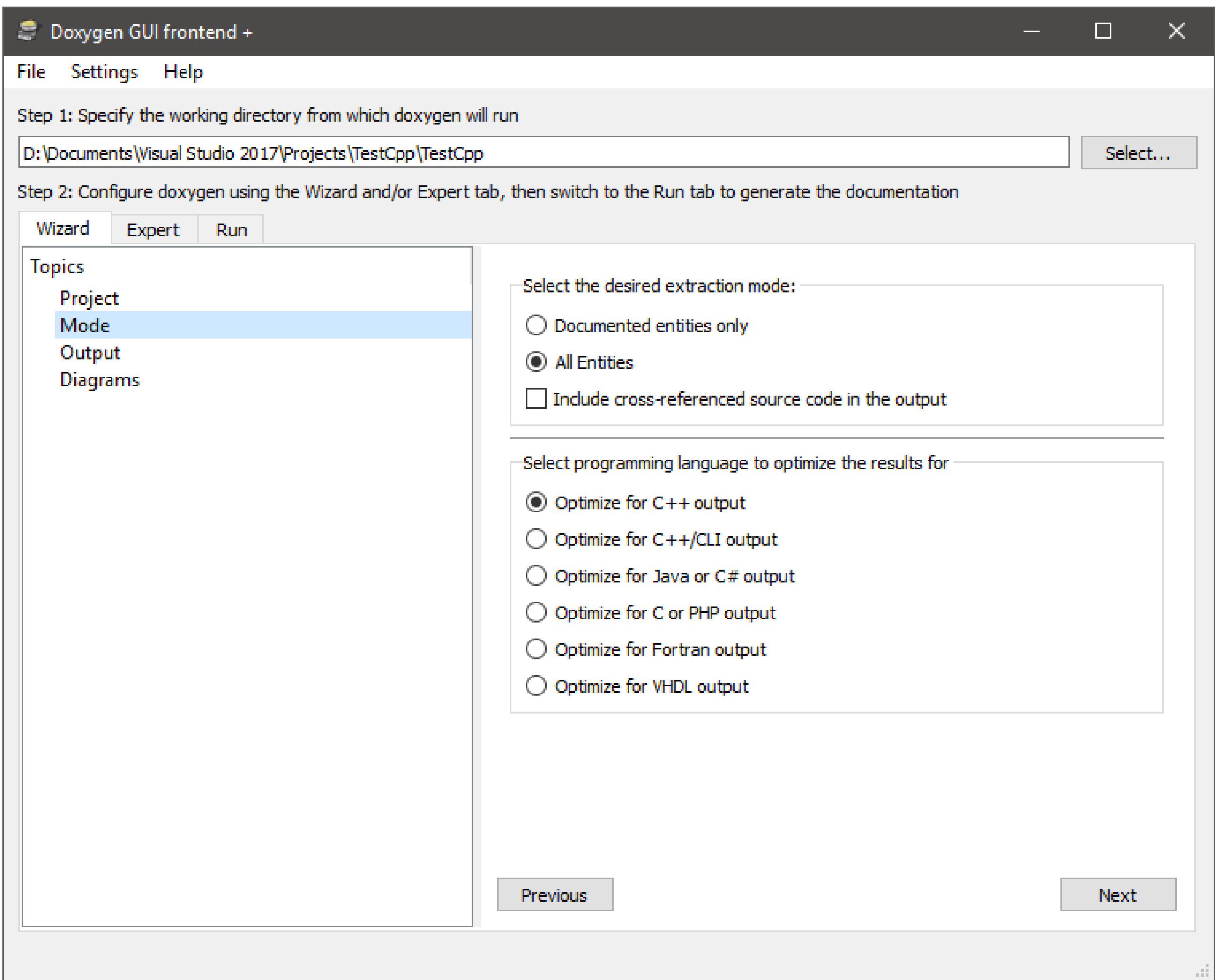


FIGURE 25.2. – Étape « Mode »

La deuxième étape est celle nommée « ***Mode*** », où l'on va indiquer que c'est du C++ (sélectionner *Optimize for C++ output*) et qu'on veut tout extraire (sélectionner *All Entities*).

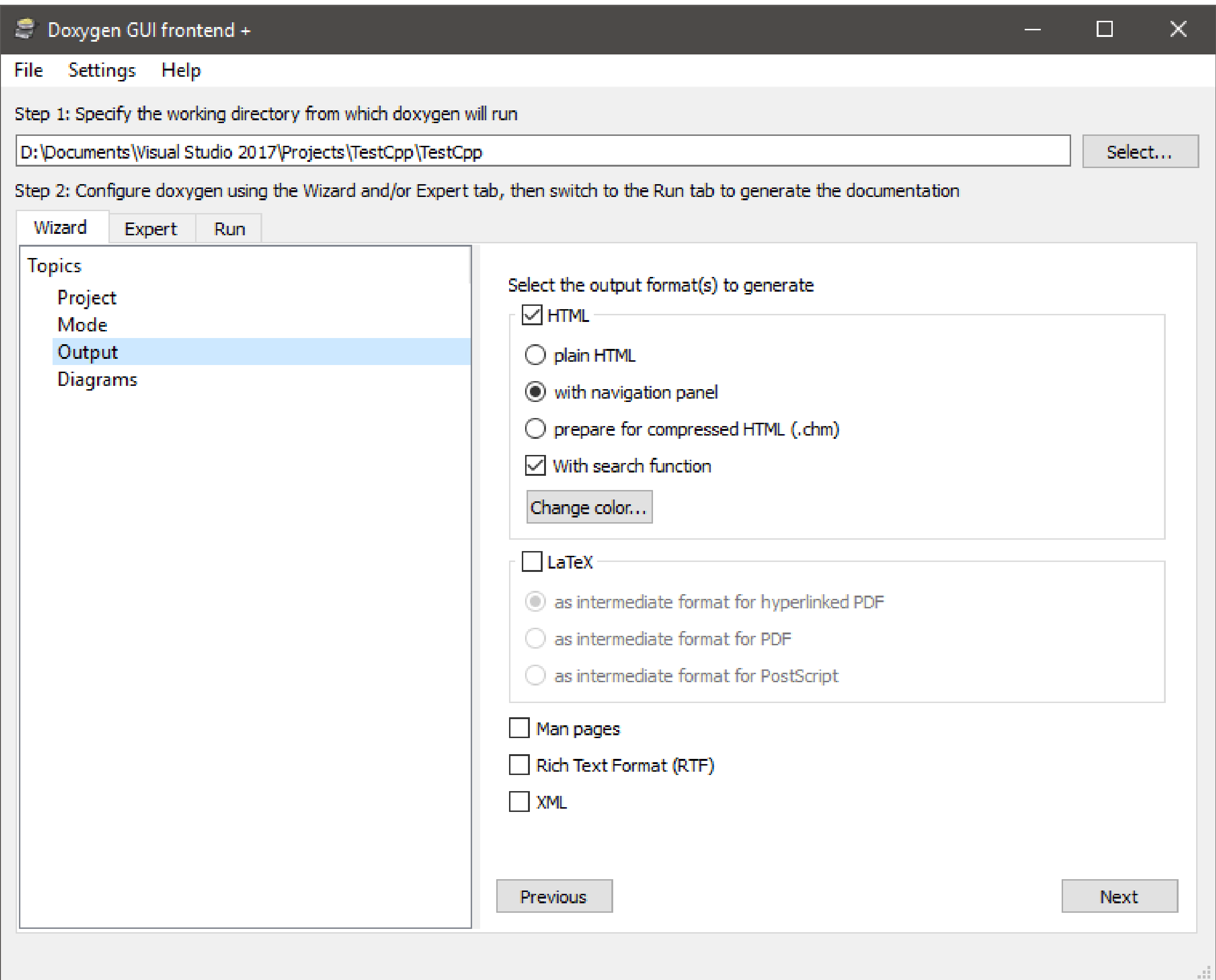


FIGURE 25.3. – Étape « Output »

Dans la troisième étape, « ***Output*** », ne cochez que **HTML** et sélectionnez *with navigation panel*, parce que les autres options de sortie ne nous intéressent pas.

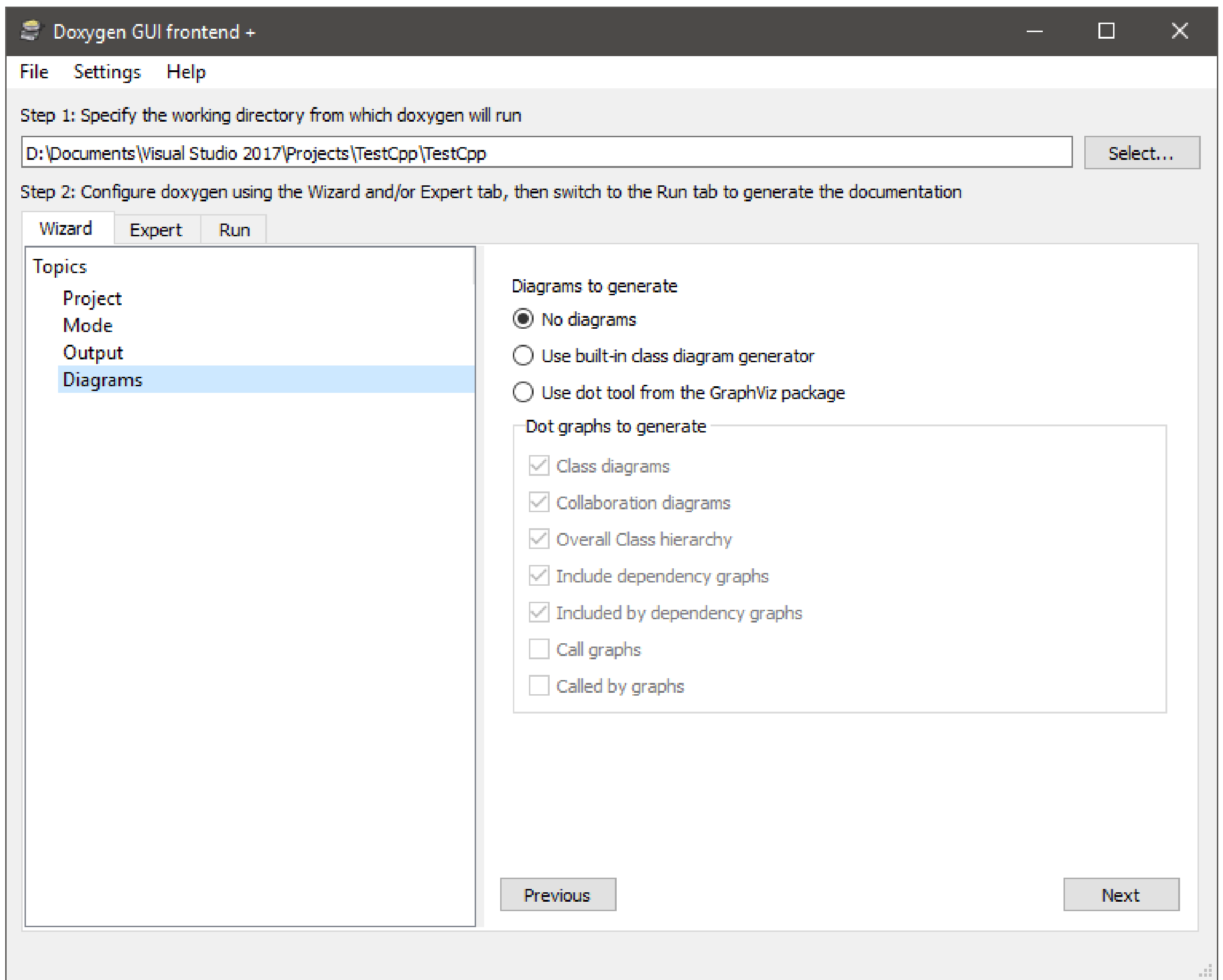


FIGURE 25.4. – Étape « Diagrams »

Enfin, dernière étape, « ***Diagrams*** », sélectionnez *No diagrams* et tout est bon. On peut passer à l'onglet « *Run* », d'où nous allons lancer la génération de la documentation en cliquant sur *Run doxygen*. Une fois cette étape faite, un dossier **html** sera présent.

> **i** Sauvegarder
>
> En quittant l'application *DoxyWizard*, on vous demandera, si vous souhaitez sauvegarder cette configuration quelque part. Quand vous relancerez l'application, il suffira de recharger cette configuration pour gagner du temps.

Par défaut, la documentation est générée en anglais, mais vous pouvez mettre la langue de votre choix en cliquant sur **Expert**, puis **Project**. Là, mettez l'option **OUTPUT_LANGUAGE** à la valeur que vous décidez, `French` pour le français.

25.2.2. Écrire la documentation

Mais concrètement, comment on écrit la documentation ? En utilisant des **balises spéciales**, que Doxygen comprendra et qu'il traitera de façon appropriée. Ainsi, on a des attributs spéciaux

pour décrire les arguments d'une fonction, sa valeur de retour, la présence de *bugs*, etc. La liste est longue, donc je n'en décrirai que certains. Libre à vous d'explorer la documentation de Doxygen pour découvrir les autres.

À noter également qu'on peut utiliser le Markdown[4] dans les blocs de commentaire, pour entourer un morceau de code, mettre en gras, etc.

Comment fait-on ? Plusieurs façons de faire sont possibles ; j'ai fais le choix de vous en montrer deux. D'abord, soit la forme `/** Commentaire */`, soit la forme `/// Commentaire`. On retrouve ici un parallèle avec les commentaires `/* */` et `//`. C'est une question de choix. Personnellement, quand le commentaire fait plus d'une ligne, j'utilise la forme longue `/** */`, sinon les trois slashs `///`.

25.2.2.1. Description d'une fonction

Et si nous reprenions notre fonction d'entrée sécurisée pour illustrer cette partie ? Je vous remets ci-dessous la dernière version, utilisant les *templates*.

Une des premières choses que nous voulons faire, c'est **d'expliquer ce que fait la fonction**, la raison de son existence. Pour cela, Doxygen fournit deux attributs.

— `@brief` sert à **décrire brièvement** la fonction. En une ou deux phrases, on décrit la fonction et ce qu'elle fait.
— `@details` sert à **décrire en détails** la fonction. Ici, on entre dans les détails, en expliquant ce qu'elle fait, ce qu'elle ne fait pas, les exceptions qu'elle peut lever potentiellement, etc.

Appliquons donc ce que nous venons d'apprendre à notre code. J'ai écrit un certain texte, mais vous pouvez très bien mettre le vôtre.

Si vous naviguez dans les fichiers HTML créés, vous allez trouver, dans la documentation du fichier C++ où se trouve la fonction (`main.cpp` par exemple), un encart présentant la fonction avec sa signature, une description courte puis une plus détaillée.

25.2.2.2. Les paramètres

Maintenant, attaquons un autre morceau très important, à savoir **la description des paramètres**. Il est important que l'utilisateur sache ce que signifie chaque paramètre et s'il y a des préconditions à respecter. Ça tombe bien, on a des attributs spécifiques pour ça.

4. Comme sur Zeste de Savoir.

— `@param[type] paramètre` permet de **décrire un paramètre** de manière générale. Le type entre crochet peut prendre trois valeurs : `in` si le paramètre est une valeur lue mais non modifiée par la fonction, `out` si le paramètre est modifié par la fonction et `in,out` dans le cas des deux possibilités en même temps.
— `@pre` décrit **une précondition par ligne**. Si plusieurs attributs `@pre` sont présents les uns après les autres, ils sont regroupés dans un même bloc à la sortie.
— `@post` décrit **une postcondition par ligne**. Si plusieurs attributs `@post` sont présents les uns après les autres, ils sont regroupés dans un même bloc à la sortie.
— Parce que les *templates* n'échappent pas à la règle, `@tparam paramètre` permet de faire la même opération pour nos chers patrons.

Ceci appliqué à notre code, on obtient quelque chose comme ci-dessous. Pareil, le texte que vous avez mis peut varier.

25.2.2.3. Les exceptions

Notre fonction peut lever des exceptions, que ce soit parce que nous en lançons nous-mêmes explicitement, soit parce qu'une des fonctions appelées lance une exception en interne. Il est donc de bon ton de décrire ce qui peut potentiellement arriver. Bien sûr, être totalement exhaustif est impossible. Mais **décrire les exceptions que nous lançons explicitement** est une bonne habitude à prendre. Cela se fait très simplement avec l'attribut `@exception exception_lancée`.

25.2.2.4. Le type de retour

Nous n'en avons pas dans notre cas, car la fonction `entree_securisee` ne retourne rien. Certains aiment utiliser l'attribut `@returns` même dans ce cas, d'autres jugent que c'est inutile. Je vous laisse choisir. Voici néanmoins un exemple d'utilisation de cet attribut, avec une fonction que nous avions écrite dans le chapitre consacré (et légèrement modifiée).

25.2.2.5. D'autres attributs utiles

Terminons cet aperçu des attributs de documentation d'une fonction en en présentant quelques-uns qui permettent de signaler des informations supplémentaires, ne rentrant pas dans le cadre des attributs vus plus haut. Il arrive, par exemple, qu'on veuille signaler la présence d'un *bug* encore non corrigé ou rarissime, d'une modification ou amélioration prévue mais pas encore faite, d'un avertissement concernant un aspect particulier du code, etc.

— `@bug` sert à **décrire la présence d'un *bug*** dans le code. Cet attribut aurait pu, par exemple, être utilisé dans les anciens T.P où nous ne pensions pas à prendre en compte la fermeture potentielle du flux. On aurait pu ajouter un commentaire du type `@bug Si le flux est fermé, la fonction rentre dans une boucle infinie`.
— `@todo` sert à **décrire une action à faire plus tard**. On pourrait s'en servir pour se rappeler, dans notre T.P, de mettre à jour le code pour utiliser un flux d'entrée générique à la place de `std::cin`.
— `@note` permet d'**ajouter des informations génériques**. Si votre code implémente une fonction mathématique, vous aurez peut-être envie de l'écrire clairement. Ce bloc est tout à fait adapté à ça.
— `@warning` permet d'**ajouter des avertissements**. On peut vouloir prévenir, par exemple, qu'une fonction ne doit pas être utilisée en même temps qu'une autre, où bien qu'un paramètre doit être d'abord correctement initialisé sous peine de voir le programme planter.

25.2.2.6. Les structures et les énumérations

Bien évidemment, il ne faut pas oublier de commenter nos types personnalisés. Non seulement le type lui-même, mais chacun des champs. Pour cela, nous disposons des attributs `@struct` et `@enum`. Chacun fonctionne sur le même principe.

— D'abord, on écrit **l'attribut puis le nom du type**.
— Ensuite, une **description brève**.
— Enfin, une **description plus détaillée**.

De même, chaque champ peut faire l'objet de description courte, détaillée, d'avertissements, etc. En général, seul une description succincte est nécessaire. Examinez par vous-mêmes un exemple de structure et d'énumération documentées.

25.3. Quelques bonnes pratiques

Vous savez maintenant documenter votre code et c'est une excellente chose. Une bonne documentation fait gagner du temps et est la meilleure alliée d'un développeur découvrant un nouveau code. Mais doit-on tout documenter ? Absolument tout ? **Y a t-il des bonnes pratiques** ?

25.3.1. Ce qu'il faut documenter

Dans l'idéal, **toutes vos fonctions devraient être documentées**. Utilisez Doxygen pour détailler vos fonctions, leur utilisation, ce qu'elles attendent, les exceptions qu'elles peuvent lancer, etc. C'est assez fastidieux, je vous l'accorde. Mais ne pas le faire est une très mauvaise pratique. Qui n'a jamais pesté contre un appareil qu'il a dû utiliser sans manuel ni instructions ? Si quelqu'un vient à utiliser votre code, il n'aimera pas non plus avoir à deviner ce que celui-ci fait, les exceptions qu'il lance, etc.

Mais si je suis seul, c'est pas un peu inutile ?

Cela vous servira même si vous êtes seul. Ne croyez pas que vous vous souviendrez de ce que font vos 4000 lignes de code quand vous rouvrirez le projet après plusieurs mois d'absence s'il n'y a pas de documentation. Ce qui vous semble clair maintenant ne le sera peut-être pas plus tard. Épargnez-vous des problèmes, documentez. Pour ne pas vous démotiver face à l'immensité de cette tâche, commencez tôt et faites en petit à petit. Ainsi, tandis que votre base de code augmentera, elle sera bien documentée et facile à comprendre.

25.3.2. Les fichiers d'en-tête ou source ?

Nous n'aimons pas la recopie. C'est une source d'erreurs et d'incompréhensions. Imaginez que la documentation d'un prototype dans un fichier d'en-tête soit légèrement différente de celle de son implémentation dans le fichier source. Qui a raison ? Qui est à jour ? Un vrai nœud à problèmes que de maintenir deux fois la même documentation à jour.

Heureusement, Doxygen est assez intelligent et affiche la documentation même si on demande à voir celle du fichier source. Il comprend qu'on a affaire aux mêmes fonctions, avec d'un côté le prototype et de l'autre l'implémentation.

25.3.3. Prenons un exemple

Parfois, certaines fonctions sont assez compliquées à utiliser et avoir un exemple fonctionnel sous les yeux nous aide vraiment. Vous-mêmes l'avez certainement remarqué en vous plongeant dans la documentation C++. **Un exemple, ça aide**. Voilà pourquoi j'ai choisi de séparer cette section du reste de la présentation de Doxygen, pour que vous compreniez bien que c'est important.

Une façon de faire consiste à utiliser l'attribut `@note`, pour bien différencier des autres sections, puis à écrire un petit exemple entre les attributs `@code` et `@endcode`.

```
```

25.3.4. Commentaire vs documentation

La découverte et l'utilisation de Doxygen signifie t-elle la fin des commentaires comme nous avions appris dès le chapitre introductif à C++ ? Bien sûr que non. En fait, les deux ne sont pas opposés, simplement, **les commentaires ne servent pas les mêmes buts**.

La documentation a pour but d'être, si possible, **exhaustive** et de **décrire le quoi, le comment**. On s'en sert pour expliquer ce que fait une fonction, les paramètres qu'elle attend, ce qu'elle renvoie, les exceptions qu'elle peut lever, ses contrats, etc. On s'en sert aussi pour illustrer le fonctionnement du code par des exemples, comme vu plus haut. **Si la documentation**

est bien faite, un développeur n'aura jamais besoin d'aller lire le code source pour comprendre.

Les commentaires eux, sont plutôt à **usage interne** et **expliquent le pourquoi**. On s'en sert pour **expliciter ce que le code ne dit pas**, pour décrire les raisons qui ont poussé à faire ou ne pas faire telle ou telle chose. Ils seront donc moins présents certes, mais pas moins utiles.

25.3.4.1. De bons commentaires...

Par exemple, on peut vouloir expliquer le choix d'une structure de données qui sort de l'ordinaire. Vous rappelez-vous de `std::map` et `std::unordered_map` ? La deuxième est souvent choisie pour des raisons de performances. Ainsi, vous voudrez peut-être expliquer pourquoi vous avez choisi la première avec un bref commentaire.

On peut aussi s'en servir pour justifier un code un peu sale ou exotique.

De la même manière, si vous voulez tester si un nombre N est premier ou pas, vous pouvez appliquer une astuce consistant à ne tester que de 2 à $\sqrt{N}$. Cela peut faire l'objet d'un commentaire explicatif.

Ces commentaires sont utiles, car seulement en lisant le code, vous n'auriez pas compris **pourquoi** il a été écrit de cette façon.

25.3.4.2. ...et de mauvais commentaires

Malheureusement, ce qu'on voit le plus souvent, ce sont les mauvais commentaires. Que ce soit parce qu'ils se contentent de redire ce que dit déjà le code, parce que le code est mal écrit, ou bien encore parce qu'ils ne sont pas clairs, le fait est **qu'ils n'apportent rien de bon**. Un exemple ? La plupart des commentaires écrits dans ce cours répètent ce que le code dit. Bien sûr, dans votre cas, c'est nécessaire car vous êtes en plein apprentissage. Par contre, pour des développeurs chevronnés, ce genre de commentaires est inutile.

Un autre type de commentaire problématique, c'est celui qui explique le code parce que celui-ci n'est pas clair. Regardez la façon dont cela est illustré dans l'exemple suivant.

Le code pourrait être amélioré en utilisant une constante, ce qui supprimerait le commentaire au passage.

Dans le même genre, les commentaires signalant une accolade fermante, ceux signalant la date de dernière modification, ceux retraçant l'historique des modifications d'un fichier, tout ça fait partie des commentaires qui n'ont pas lieu d'être.

> **Ressources supplémentaires**
>
> La distinction peut vous sembler dur à comprendre et la frontière entre un bon et un mauvais commentaire peut paraître bien mince. Une littérature informatique abondante aborde ce sujet, que nous ne pouvons, dans le cadre de ce cours, parcourir exhaustivement. Si vous souhaitez aller plus loin, un chapitre du livre Coder proprement y est dédié.

25.3.5. En résumé

- La documentation est une des meilleures amies du développeur.
- La documentation C++ est classée en divers thèmes, allant du langage lui-même aux algorithmes, en passant par `std::string` et les fonctions mathématiques.
- On retrouve souvent les mêmes sections, décrivant les différents prototypes, les paramètres, les exceptions possibles, ainsi que des exemples concrets.
- Doxygen nous permet d'écrire la documentation de nos programmes. C'est une bonne habitude à prendre.
- Certaines bonnes pratiques peuvent nous aider à écrire des commentaires et de la documentation de qualité.

Contenu masqué

Contenu masqué n°66 : Correction

Puisqu'on parle de chaînes de caractères, on va donc chercher du côté de `basic_string` . On remarque, dans la liste des fonctions disponibles, une certaine `replace` . Le premier prototype est justement celui qui répond à l'énoncé. On obtient donc ce qui suit.

Retourner au texte.

Contenu masqué n°67 : Indice

Il faut chercher dans les fonctions mathématiques . Retourner au texte.

Contenu masqué n°68 : Correction

En parcourant la liste des fonctions contenues sur cette page de documentation, on note le triplet `hypot`, `hypotf` et `hypotl`. La description associée dit que ces fonctions calculent la racine carrée de la somme des carrées de deux nombres, soit $\sqrt{x^2 + y^2}$. Bien que, de par son nom, cette fonction semble plutôt destinée à calculer une hypoténuse, elle se révèle quand même adaptée. La correction est toute simple.

Retourner au texte.

Contenu masqué n°69 : Indice

Vous devez chercher du côté de la page complex . Retourner au texte.

Contenu masqué n°70 : Correction

Alors, on arrive sur la fameuse page en question, où l'on découvre un nouveau type, `complex`. C'est un *template*, donc on va devoir lui passer un type, en l'occurrence `double`.

On voit ensuite, plus bas, la fonction `sqrt` , qui répond à la première partie de l'exercice. C'est une fonction non-membre, donc qui s'utilise en se préfixant de `std::`.

Plus haut, dans les fonctions membres, on voit `real` et `imag` , qui donnent respectivement la partie réelle et la partie imaginaire de notre résultat complexe. On a donc tout ce qu'il nous faut.

Retourner au texte.

Contenu masqué n°71 : Correction

Qui dit algorithme dit bien souvent le fichier correspondant `<algorithm>` . C'est effectivement dedans que nous allons trouver ce qu'il nous faut et l'algorithme en question se nomme `std::transform` . Avez-vous écrit un code comme celui-ci ?

```

```

Si oui, alors vous n'avez pas bien lu la documentation. En effet, lancez ce programme et il plantera. Pourquoi ? Parce que la chaîne de sortie est vide, donc on ne peut pas modifier les éléments d'une chaîne vide puisqu'il n'y en a pas. La documentation apporte une précision.

> `std::transform` applies the given function to a range and stores the result in another range, beginning at `d_first`.
>
> *Documentation de `std::transform`.*

Il faut que le conteneur de sortie ait la même taille que celui d'entrée. Ou, tout du moins, en théorie. Si vous lisez le code d'exemple situé en fin de page, vous allez remarquer la présence d'une fonction inconnue jusque-là : `std::back_inserter`. En lisant sa documentation , on apprend qu'elle créée un `std::back_insert_iterator` .

Qu'à cela ne tienne, allons lire la documentation de ce nouveau genre d'itérateur. Celle-ci nous donne une information très intéressante.

> `std::back_insert_iterator` is an `LegacyOutputIterator` that appends to a container for which it was constructed. The container's `push_back()` member function is called whenever the iterator (whether dereferenced or not) is assigned to.
>
> *Documentation de `std::back_insert_iterator`.*

Traduit, cela signifie qu'à chaque fois que l'itérateur est modifié, la fonction `push_back` du conteneur en question est appelée. C'est exactement ce qu'il nous faut pour notre conteneur de sortie vide. Voici donc le code final.

```

```

Cet exercice vous a semblé plus difficile ? C'était le cas. Quand on ne connait pas quelque chose, bien souvent, une seule page de documentation n'est pas suffisant. Il faut parfois faire des recherches complémentaires. Heureusement pour vous, la documentation C++ présente sur ce site est accompagnée d'exemples qui vous aideront à bien saisir les difficultés et particularités potentielles. Retourner au texte.

26. Compilation en cours…

Depuis le début de ce tutoriel, nous avons compilé et recompilé de nombreuses fois nos programmes. Vous savez, depuis l'introduction, que la compilation transforme notre code source en un programme exécutable qui sera compris par l'ordinateur. Mais, concrètement, que se passe-t-il pendant la compilation ? Quelles en sont les différentes étapes ?

Le but de ce chapitre va être de **présenter en détails la compilation d'un code C++**.

26.1. Le préprocesseur

La toute première étape est celle du **préprocesseur**. Ce nom vous est familier ? Normal, on fait déjà appel à lui pour chaque ligne qui commence par `#`.

26.1.1. Inclure des fichiers

Dans le cas de la directive `#include`, il s'agit d'inclure tout le fichier demandé. Il est possible de voir à quoi ressemble notre code après son passage.

— Avec **Visual Studio**, il faut faire un clic-droit sur le nom du projet et cliquer sur Propriétés. Là, on va dans Propriétés de configuration -> C/C + -> Préprocesseur, puis on met le champ Prétraiter dans un fichier à Oui (/P). Attention cependant, **cette option bloque la suite de la compilation**. Donc désactivez la sortie du préprocesseur une fois que vous avez fini d'explorer cette option.

— Avec **QtCreator**, il faut modifier le fichier **.pro** en lui ajoutant la ligne `QMAKE_CXXFLAGS += -save-temps`. Le fichier obtenu après traitement a une extension en `.ii`.

— Pour ceux qui travaillent en ligne de commande, pour GCC c'est `g++ -E fichier.cpp -o fichier.i` et pour Clang, `clang++ -E fichier.cpp -o fichier.i`.

Le résultat n'est pas le même en fonction du compilateur, mais **le fichier devient conséquent** après le passage du préprocesseur. Pour un simple `#include <iostream>`, j'obtiens plus de 17.800 lignes pour Clang, plus de 18.000 lignes avec GCC et même plus de 50.000 lignes avec Visual Studio (dont pas mal vides, certes) !

26.1.2. Conditions

Mais le préprocesseur ne se contente pas d'inclure bêtement ce qu'on lui demande et c'est tout. Il est capable d'exécuter d'autres instructions, notamment celles permettant d'éviter les inclusions multiples. Ce sont des **instructions conditionnelles** pour le préprocesseur. Je vous remets un exemple ci-dessous.

La première ligne teste **si une constante de préprocesseur `TEST_HPP` existe**. Si ce n'est pas le cas, alors on exécute le code qui suit. Justement, la ligne d'après définit la constante `TEST_HPP`. Ainsi, si l'on tente d'inclure de nouveau le fichier, la première condition devient fausse et rien n'est inclus. Enfin, l'instruction `#endif` permet de fermer la portée du `#ifndef`, tout comme les accolades fermantes pour `if`.

Ce mécanisme de conditions est très utilisé au sein des fichiers d'en-tête de la bibliothèque standard. En effet, cela permet d'**adapter le code en fonction de la version du compilateur ou de la norme C++**, entre autres. Par exemple, la bibliothèque standard fournie avec Visual Studio utilise le code suivant pour ne définir la fonction `std::for_each_n` que si l'on compile en C++17.

26.1.3. *Debug* ou *release* ?

Le préprocesseur est également utilisé pour **modifier la bibliothèque standard selon qu'on compile en *debug* ou en *release***. Il s'agit de modes de compilation qui sont utilisés en fonction du moment.

26.1.3.1. *Debug* — Quand on code

Le mode ***debug*** inclut, dans le code de la bibliothèque standard, de **nombreuses vérifications et autres tests**. C'est le mode dans lequel est compilé le code quand on est encore en plein développement, qu'on cherche à corriger des *bugs* ou qu'on fait des tests. Depuis le début du cours, nous ne compilons qu'en mode *debug*. C'est pour cette raison que, quand nous utilisons un indice trop grand, le programme plante avec un message d'erreur.

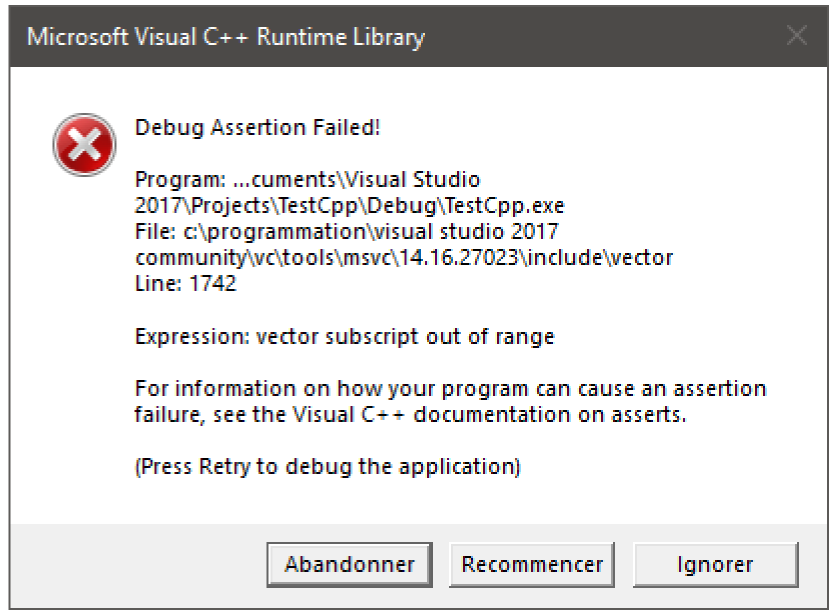


Figure 26.1. – Erreur affichée en mode debug.

Si on regarde le code de `std::vector` tel qu'il est fourni avec Visual Studio, on trouve les lignes suivantes. On retrouve le message qu'on voit dans la fenêtre ci-dessus.

La macro `_ITERATOR_DEBUG_LEVEL` est elle-même définie uniquement si une macro `_HAS_ITERATOR_DEBUGGING` est définie comme valant autre chose que `0`. Cette dernière dépend de l'existence de la macro `_DEBUG`, qui est uniquement définie quand on compile en mode *debug*.

26.1.3.2. *Release* — Quand on distribue

Le mode ***release***, lui, **se débarrasse de toutes ces vérifications**, **optimise le code** pour le rendre plus rapide à exécuter et **réduit la taille de l'exécutable** qu'on obtient en sortie. On compile avec ce mode quand le programme est prêt et qu'on veut le distribuer à d'autres personnes. À ce stade, on juge que le programme est suffisamment bon et correctement testé. Il est donc inutile de continuer à inclure tous les tests et autres vérifications, qui rendent le code plus gros et plus lent.

Typiquement, **les assertions sont désactivées en *release***. Bien souvent, `assert` n'est pas implémentée en tant que fonction, comme nous en avions l'habitude, mais à l'aide du préprocesseur, ce qui permet de la supprimer du code, ou de la rendre inopérante, quand on passe en *release*.

26.2. La compilation

Maintenant que le préprocesseur a fini son travail, place à la compilation à proprement parler. Chaque fichier que le préprocesseur fournit est traité individuellement. Le but de cette phase est de **transformer le C++ en code exécutable** par l'ordinateur.

C'est lors de cette étape que **beaucoup de vérifications ont lieu** et donc d'erreurs. Par exemple, dans le cas où le compilateur ne trouve pas la déclaration d'une fonction que vous appelez, il décidera de faire s'arrêter la compilation. Vous aurez alors un message d'erreur vous indiquant ce qui n'a pas fonctionné.

26.2.1. Les templates

Comme dit dans le chapitre correspondant, le compilateur va **instancier chaque *template*** qu'il trouve avec les bons types. C'est pendant cette étape que des erreurs incompréhensibles de *templates* peuvent arriver, par exemple quand on tente d'appliquer `std::sort` sur une `std::list`.

Prenez l'exemple suivant d'un code C++ très simple.

Avec Clang, il est possible d'utiliser les options `-Xclang -ast-print -fsyntax-only` pour afficher le code après l'instanciation des templates. Avec GCC, c'est la commande `-fdump-tree-original` qui génère des fichiers **.original** qu'on peut afficher.

On voit dans le résultat suivant, obtenu avec Clang, que le *template* est instancié deux fois, avec `int` et avec `double`.

26.2.2. `constexpr`

Nous avions dit, dans le chapitre sur la documentation, que nous reviendrons sur ce mot-clé en temps voulu. C'est maintenant le moment d'en parler, car il a un rapport avec la compilation. En effet, il indique au compilateur qu'**une variable ou une fonction peut déjà être évaluée pendant la compilation** et non plus seulement à l'exécution du programme. L'avantage, c'est que c'est le compilateur qui effectue le travaille une fois pour toutes, ce qui rend **l'exécution du programme plus rapide**.

26.2.2.1. Des constantes

En déclarant des constantes avec le mot-clef `constexpr`, on les rend utilisables à la compilation. Typiquement, dans le cas où l'on fixe d'avance la hauteur et la largeur d'une matrice, utiliser `std::array` est tout à fait possible.

`constexpr` et `const`

Implicitement, utiliser `constexpr` sur une variable revient à la déclarer `const`, ce qui est logique. Personnellement, dans le but d'être explicite, je précise également `const`, mais ce n'est pas une obligation, simplement une question de goût.

Par contre, pour que l'instruction soit valide et que le programme puisse compiler, il faut qu'elle soit **évaluable à la compilation**. Cela signifie que toute l'expression doit être comprise et utilisable par le compilateur pendant la compilation et que rien ne doit dépendre de variables non-constantes, d'entrées/sorties, etc.

```
```

26.2.2.2. Des fonctions

Grâce au mot-clé `constexpr`, nous somme capables d'écrire des fonctions qui s'exécuteront **pendant la compilation**, ce dont nous étions pas capables jusque-là, tout du moins sans utiliser les *templates*. En plus, la syntaxe est très simple, puisqu'il suffit d'ajouter `constexpr` au prototype de la fonction.

```
```

Cela a néanmoins un effet un peu particulier.

— Si la fonction est appelée avec des arguments `constexpr`, comme des littéraux ou des variables `constexpr`, alors **elle est exécutée à la compilation**.
— Dans le cas contraire, avec des arguments non-`constexpr`, elle se comporte normalement et est exécutée **pendant l'exécution du programme**.

Prenons un exemple en calculant la factorielle d'un nombre entier. C'est une opération mathématique simple, qui se définit comme le produit de tous les entiers de 1 à n. La formule est la suivante.

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times ... \times n$$

Ainsi, 2! vaut 2, mais 10! vaut déjà 3628800 ! C'est donc une opération qu'il convient de faire calculer au compilateur si cela est possible. En plus, son implémentation est toute simple, comme vous le montre le code suivant. Notez que dans le premier cas, la fonction est appelée pendant la compilation, alors que dans le deuxième cas, pendant l'exécution.

```
```

?

Alors pourquoi ne pas déclarer toutes nos fonctions `constexpr` ? Comme ça on gagne en performances.

Parce que `constexpr` est **soumis à des restrictions**. En C++11, il ne pouvait pas y avoir de boucles ou de `if / else`. Depuis C++14, ces restrictions ont été levées, ce qui permet l'implémentation de la factorielle telle que vous la voyez plus haut.

Et sans parler de ces considérations, une fonction ne peut être `constexpr` que si toutes ses instructions sont évaluables pendant la compilation. Cela nous interdit donc d'utiliser les entrées/sorties ou des fonctions qui ne sont pas `constexpr`. Voilà pourquoi toutes nos fonctions ne peuvent pas être exécutées pendant la compilation.

26.2.3. La compilation à proprement parler

Maintenant, il s'agit de transformer tout ce code C++ en quelque chose de compréhensible par l'ordinateur. Lui ne comprend que les 0 et les 1. Il n'a que faire de nos noms de variables, de fonctions ou de type. La transformation du C++ en code machine se fait en plusieurs étapes. On peut en citer quelques-unes.

— **L'analyse lexicale**, où il déduit que `auto` ou `int` sont des mots-clés, `main` ou `x` sont des identificateurs, `=` ou `+` sont des opérateurs, etc.
— **L'analyse syntaxique**, où le compilateur regarde si la syntaxe fait sens et ce qu'elle signifie. Ainsi, écrire `auto = 5` sera rejeté pendant la compilation car cette syntaxe n'a aucun sens.
— **Les optimisations**, notamment en mode *release*. Cette étape supprime du code inutile, modifie certaines portions pour qu'elles s'exécutent plus vite, etc.
— **Génération du code pour la plateforme cible**. Le fichier objet ne sera pas pareil selon qu'on compile sous Windows ou GNU/Linux, donc le compilateur fait des adaptations pour la machine cible.

Note

Je ne détaille pas plus la compilation du C++ à proprement parler car c'est **affreusement compliqué**. C++ a la réputation d'être un langage dont l'écriture du compilateur est très compliquée et cette réputation n'est pas volée.

Le compilateur va au final générer des **fichiers objets**, un pour chaque fichier **.cpp** qu'il trouve, comprenant son code propre et tous les fichiers d'en-têtes inclus.

Un fichier objet, reconnaissable à son extension en **.o** ou **.obj**, n'est rien d'autre que **du code machine** mais seul, **sans aucun lien avec le reste du monde**. Tel quel, **il n'est pas exécutable**, car il lui manque des informations. Pour cela, il faut utiliser le **linker**, ce que nous allons voir dans la partie suivante.

Dans le cas de Visual Studio, ils sont présents dans le dossier de sortie. Dans le cas de GCC et Clang, il faut utiliser l'option `-c` lors de la compilation.

26.2.4. Une étape intermédiaire cachée

Beaucoup de compilateurs génèrent maintenant directement des fichiers objets, rendant invisible une étape intermédiaire, la **transformation de C++ en** [**Assembleur**] , un langage de bas-niveau très proche du langage machine. Ce langage était ensuite passé à un **assembleur** [5], qui s'occupait de transformer ce code en fichier objet.

Il est cependant possible d'obtenir le code Assembleur généré avec la bonne option.

— Avec Visual Studio, il faut aller dans les propriétés du projet, puis dans C/C + -> Fichiers de sortie. Là, il faut mettre l'option Sortie de l'assembleur à **Assembleur, code machine et source (/FAcs)**.
— Avec Qt Creator, si vous avez déjà ajouté la ligne `QMAKE_CXXFLAGS += -save-temps` à votre fichier **.pro**, vous n'avez rien d'autre à faire. Le fichier assembleur est celui avec une extension **.s**.
— Avec GCC et Clang, il faut compiler avec l'option `-S`.

Attention, il s'agit d'un langage très… brut de décoffrage. Si vous ne me croyez pas, jetez un œil par vous-mêmes, vous allez être surpris.

26.2.5. Influer sur la compilation

Il est possible de passer d'autres informations au compilateur, notamment pour lui demander d'effectuer certaines opérations ou des tests supplémentaires. Cette possibilité est importante puisque plus le compilateur fait de vérifications, **plus il peut nous signaler d'éventuels problèmes** dans notre code. C'est ce qu'on appelle les *warnings*. Loin de nous embêter, le compilateur est donc un précieux allié.

26.2.5.1. Un mot sur les outils

Comme le compilateur fourni avec Visual Studio est différent de GCC ou de Clang, les options de compilation et les *warnings* sont aussi différents et ne s'active pas forcément de la même façon.

Avec Visual Studio, allez dans les propriétés du projet, puis C/C + -> Général et là, mettez le champ Niveau d'avertissement à la valeur Niveau 4 (/W4). Cela augmente le nombre de vérifications effectuées et donc de *warnings* générés. Je ne conseille pas (/Wall) parce que cette option est si sévère qu'elle va même générer des avertissements dans le code de la bibliothèque standard.

Avec GCC ou Clang, il faut **rajouter des options** supplémentaires, qu'on appelle couramment des *flags*. Il en existe énormément, tant pour GCC que pour Clang. Mais les deux plus

5. Oui, il s'agit du même nom pour le langage et pour l'outil qui compile ce langage. On les différencie toutefois en employant un « A » majuscule pour parler du langage.

utilisés sont `-Wall` et `-Wextra`, qui activent en sous-main de nombreux *warnings*, ceux revenant le plus fréquemment.

> **Un mot sur les *warnings***
>
> Je vous conseille de toujours activer les *warnings* suggérés ci-dessus. Plus le compilateur travaille pour vous, plus vous pouvez corriger tôt d'éventuels problèmes et améliorer ainsi la qualité de votre code.

26.2.5.2. Conversion entre types

Dès le début de ce cours, nous avons vu des types simples, permettant de stocker des caractères (`char`), des entiers (`int`, `std::size_t`) et des réels (`double`). Ils sont bien pratiques, mais si on commence à les mélanger, on peut avoir des problèmes.

En effet, comme déjà signalé dans le chapitre sur les erreurs, tous n'ont pas la même taille. Un `int`, par exemple, ne peut pas stocker des nombres aussi grands que `double` peut. Ainsi, convertir un `double` en `int` entraîne **une perte d'information**. C'est un peu comme vouloir transvaser de l'eau depuis un seau dans un gobelet.

Le code suivant illustre bien le principe, puisque en passant d'un `double` à un `int`, on perd comme information la partie décimale du nombre.

Le compilateur ne va d'ailleurs pas se gêner pour vous le dire.

Le compilateur nous prévient que c'est une erreur et qu'il faut une **conversion explicite**, ou en anglais *cast*, pour résoudre le problème. Clang est le plus sympa puisqu'il propose une solution, qui consiste à utiliser le mot-clé `static_cast`. Ce dernier permet de dire explicitement au compilateur qu'on veut convertir un type vers un autre, en l'occurrence ici un `double` vers un `int`. Il s'utilise comme suit.

Une fois la conversion explicite appliquée, le code fonctionne sans problème.

Sans les avertissements du compilateur, **l'erreur serait sans doute passée inaperçue**. Si l'on comptait faire des calculs avec nos valeurs réelles, ceux-ci auraient été faussés puisque les parties décimales auraient disparues. Avec l'avertissement, nous pouvons soit corriger notre erreur, soit mettre le *cast* pour montrer que notre conversion est voulue et volontaire.

26.2.5.3. Oubli d'initialiser

Imaginez que vous oubliez d'écrire les accolades `{}` après avoir déclaré une variable. C'est moyennement cool, puisque du coup, **on ne lui donne pas de valeur par défaut**. Elle peut donc valoir tout et n'importe quoi. Le message du code suivant peut, ou non, s'afficher. C'est impossible à savoir.

Heureusement, le compilateur ne manque pas de le souligner.

26.2.5.4. Paramètres inutilisés

Imaginez que vous déclarez une fonction attendant deux paramètres, mais que vous n'en utilisez qu'un seul. C'est peut-être volontaire, parce que vous n'avez pas encore fini de l'implémenter, mais c'est peut-être une erreur, soit que vous avez oublié une portion de code à écrire, soit oublié de supprimer un paramètre désormais inutile.

Votre compilateur se demande quelle est votre intention et vous affiche donc un message.

26.2.5.5. *Release* et *debug*

Les options de compilation servent aussi à indiquer au compilateur le mode choisi. Il est ainsi capable d'optimiser le programme pour le rendre plus petit et plus rapide. Une partie de ce travail est faite quand le préprocesseur passe, mais d'autres améliorations et optimisations sont effectuées par le compilateur.

Avec Visual Studio, il faut simplement changer la valeur du cadre de l'image ci-dessous, qui se trouve près du triangle vert Débogueur Windows local.

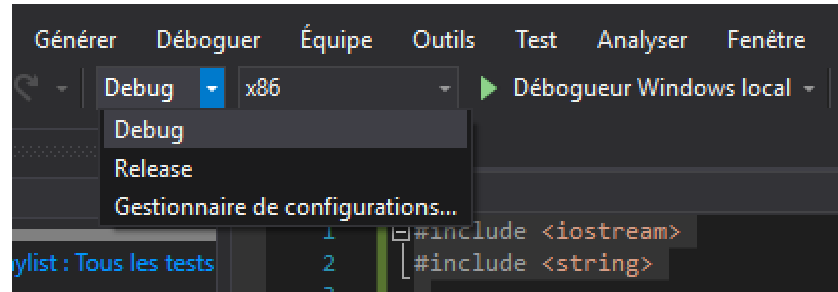


Figure 26.2. – Choix du mode de compilation avec Visual Studio 2017.

Avec GCC et Clang, c'est **l'option de compilation `-O2`** qu'il faut passer. Celle-ci cherche à optimiser le programme pour rendre son exécution plus rapide.

26.3. Le linker

> **Un mot sur les outils**
>
> Les propos ci-dessous sont illustrés sous GNU/Linux, mais le principe et la plupart des explications sont valables peu importe le système d'exploitation.

Maintenant que nous avons des fichiers objets, il faut **les lier** pour obtenir notre exécutable final. C'est justement le travail accompli pendant l'étape du *linkage*. Et pour illustrer mes explications, nous allons travailler avec le code ci-dessous, divisé en trois fichiers.

D'abord, le fichier d'en-tête `test.hpp`, qui ne contient qu'un simple prototype.

Ensuite, le fichier `test.cpp`, qui est l'implémentation du prototype déclaré précédemment.

Et enfin le fichier `main.cpp`, le point d'entrée.

On génère donc le code. Que ce soit avec GCC, Clang ou en utilisant Visual Studio, on obtient deux fichiers objets, un par fichier source.

26.3.1. Une question de symboles

Un fichier objet contient, entre autres, des **symboles**. Qu'est-ce que c'est ? Un symbole correspond **au nom d'une fonction ou d'une variable**. Les noms de toutes les fonctions et variables utilisées dans chaque fichier source sont donc présents dans les fichiers objets correspondants. On peut le voir en utilisant la commande `objdump -t fichier.o -C`.

De nombreuses informations sont affichées, qui nécessiteraient beaucoup d'explication. Notons simplement qu'on retrouve notre identificateur `fonction` sur la dernière ligne. La lettre `F` indique qu'il s'agit d'une fonction. Devant, on note la présence de `.text`, la **section** dans laquelle `fonction` se trouve.

Sans rentrer dans les détails ni transformer ce cours de C++ en cours système, il faut savoir qu'un programme est découpé en plusieurs sections. **Une section regroupe certaines informations ensemble**. Citons la section **.text** qui contient le code et **.rodata**, qui contient toutes les données non-modifiables, comme les littérales chaînes de caractères.

Dans le cas du fichier objet `test.o`, comme `fonction` désigne une fonction, donc du code exécutable, il est normal que sa section soit `.text`. Et si on faisait la même analyse sur `main.o` ?

Certaines choses ne nous surprennent pas, comme `main` désignée comme étant une fonction située dans `.text`. Par contre, d'autres lignes n'ont que pour seule information `*UND*`. On trouve plusieurs choses familières, comme `std::cout` ou encore `std::ostream::operator<<(int`.

En fait, toutes les lignes contentant `*UND*` désignent des symboles n'ayant **aucune section liée**. Ce sont des symboles qui ne sont pas définis dans notre fichier objet. Il faut donc **lier celui-ci à d'autres fichiers objets**, notamment ceux qui contiennent ce qui nous manque.

Dans le cas de la bibliothèque standard, c'est fait automatiquement par le compilateur sans avoir besoin de rien faire. Mais pour résoudre le symbole `fonction()`, il faut **lier nos deux fichiers objets**. C'est très simple, la commande suivante vous le montre. Notez que l'option `-std=c++17` est absente, car ici on ne compile rien, **on *linke*, ou assemble, deux fichiers objets**.

Maintenant, on obtient un exécutable parfaitement fonctionnel. La compilation est terminée.

Au contraire, si on tente de compiler seulement `main.o`, le *linkage* échouera avec une erreur comme celle ci-dessous.

26.4. Schéma récapitulatif

Le processus de compilation d'un programme C++ peut sembler vraiment complexe quand on démarre. C'est vrai qu'il s'agit d'un domaine avancé, mais avoir une idée globale du processus et de ce qui se passe sous le capot est important. Voici donc un schéma, car une image vaut mieux que mille mots.

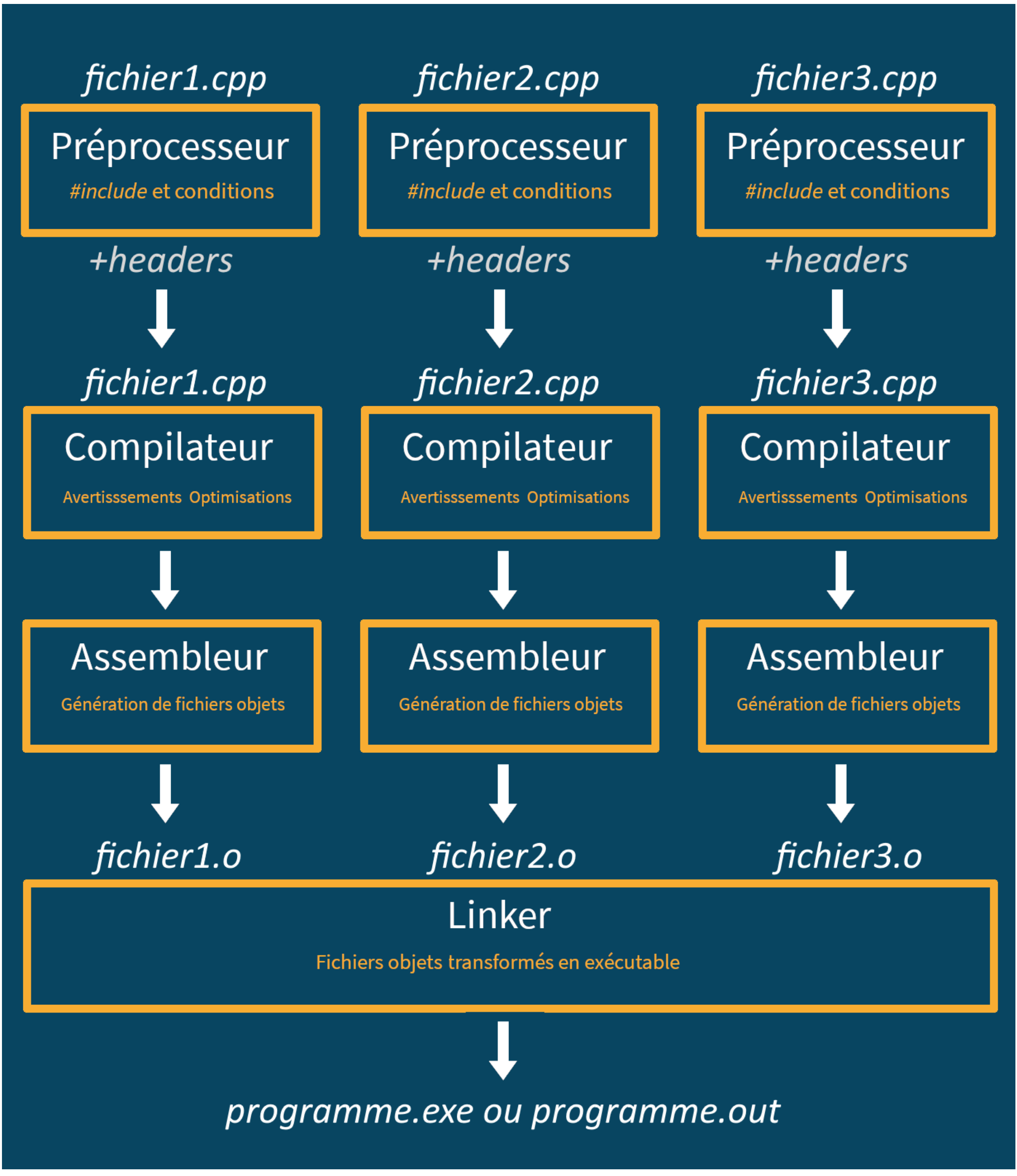


FIGURE 26.3. – Processus de compilation

26.4.1. En résumé

— Le préprocesseur commence le travail en remplaçant chaque directive `#include` par le contenu du fichier demandé et en exécutant des tests conditionnels pour activer ou désactiver certains morceaux de code.

- Le compilateur instancie les *templates*, exécute les fonctions `constexpr`, vérifie la qualité de notre code puis, en plusieurs phases, transforme le code C++ en fichier objet.
- Le *linker* fait le lien entre plusieurs fichiers objets pour résoudre des symboles manquants et achever le processus de compilation en générant un exécutable fonctionnel.
- Il n'est pas obligatoire de connaître chaque détail, mais avoir une vision globale est important.

27. Chasse aux bugs !

Malgré tout le soin qu'on peut apporter à la conception, malgré tous les *warnings* qu'on active pendant la compilation, même avec toutes les bonnes pratiques du monde, nous faisons des erreurs et nous créons des *bugs*. Il nous faut donc des outils supplémentaires pour les traquer et les éliminer.

Ce chapitre va nous faire découvrir **l'usage des débogueurs** .

27.1. Le principe

À l'exécution d'un programme, il se passe beaucoup de choses et très vite. Une fonction en appelle une autre, une variable est modifiée, une exception lancée, etc. Il est donc difficile, d'un simple coup d'œil, de savoir où est survenu le problème. Le principe du débogueur est donc de **dérouler l'exécution du programme pas-à-pas**, ce qui permet, à chaque instant, de voir où on en est et quel est l'état de nos variables et de la mémoire.

Pour ordonner au débogueur de s'arrêter à un endroit précis dans le code, on pose ce qu'on appelle un **point d'arrêt**, ou, en anglais, *breakpoint*. Chaque fois que le débogueur passe par un point d'arrêt, **le programme débogué se stoppe jusqu'à nouvel ordre** et toutes les informations courantes sont affichées, comme la liste des fonctions appelées jusque-là (la *stack trace* en anglais), la valeur de toutes les variables de la portée courante, etc.

Pour les débogueurs les plus avancés, il est également possible de modifier des données du programme, de poursuivre l'exécution à un autre endroit du code pour sauter un crash ou une exception, observer le code assembleur généré et même de déboguer des programmes à distance, par Internet notamment.

Pour faire tout ce travail, le débogueur se repose sur les **symboles de débogage**, qui sont présents quand on compile en *debug* mais souvent éliminés quand on passe en *release*. Donc, afin de pouvoir utiliser cet outil puissant, veillez à passer en mode `Debug` si vous utilisez Visual Studio, ou bien à compiler votre programme avec l'option `-g`, pour charger ces fameux symboles.

27.2. Visual Studio

Le débogueur inclus avec Visual Studio se lance très simplement en faisant F5, ou en cliquant sur l'option Débogueur Windows local. Mais il nous faut déjà un code à déboguer. Que diriez-vous de nous entrainer sur l'exemple suivant ?

Si vous lancez le code tel quel en *debug*, la console va s'ouvrir et se refermer aussitôt. Il faut donc commencer par **poser des points d'arrêt**. Pour ça, c'est très simple, on clique sur la colonne à gauche de celle indiquant les numéros de ligne. Comme une image est plus claire, voici un point d'arrêt posé.

TestCpp.cpp
TestCpp (Portée globale)

```
#include <iostream>
#include <vector>

void fonction()
{
    std::vector<int> tableau {};

    tableau.push_back(10);
    tableau.push_back(20);

    int & a { tableau.front() };
    int & b { tableau.back() };

    std::cout << "Front : " << tableau.front() << std::endl;
    std::cout << "Back : " << tableau.back() << std::endl;

    b = 5;

    std::cout << "Front : " << tableau.front() << std::endl;
    std::cout << "Back : " << tableau.back() << std::endl;
}

int main()
{
    fonction();
    return 0;
}
```

FIGURE 27.1. – Un point d'arrêt posé à la ligne 8.

27.2.1. Interface

Une fois que le programme est lancé, l'aspect de Visual Studio change un peu. Examinons ce qu'il y a de nouveau, en commençant par la petite flèche jaune, actuellement sur la ligne 8, qui indique **l'instruction à laquelle on se trouve**.

```
8    tableau.push_back(10);
9    tableau.push_back(20);
```

Figure 27.2. – On se trouve actuellement à la ligne 8.

En bas à gauche, c'est la fenêtre des variables, grâce à laquelle on peut voir, à chaque étape, **l'état des variables du programme**. Elle est divisée en trois onglets.

— **Automatique** correspond à **toutes les variables existantes jusqu'à la ligne où l'on se trouve**. Celles situées après ne sont pas encore connues donc pas encore affichées. Pour l'instant, il n'y a que `tableau`.
— **Variables locales** correspond à **toutes les variables de la portée actuelle**, existant déjà en mémoire ou non. Dans notre exemple, on retrouve donc `tableau`, ainsi que `a` et `b`, même si celles-ci n'ont pas encore de valeur mais juste une ligne « Impossible de lire la mémoire. »
— **Espion 1** permet de **garder un œil sur une variable** et d'être au courant de toutes ses modifications. On peut, par exemple, garder une trace de la taille du tableau en créant un espion `tableau.size()`.

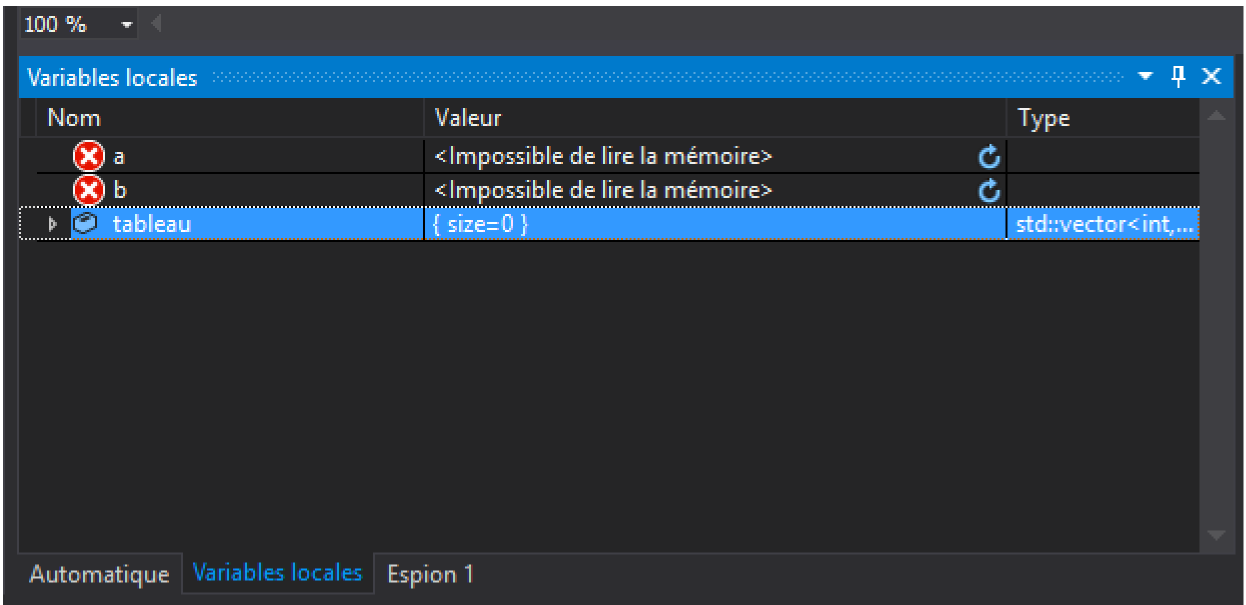


Figure 27.3. – Fenêtre des variables locales.

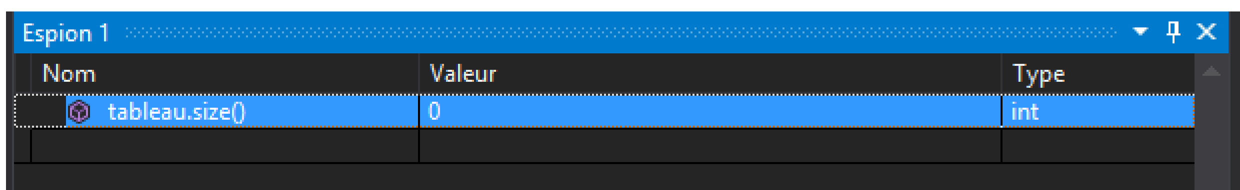


Figure 27.4. – Exemple d'espion. Quand le tableau sera agrandi ou rétréci, la valeur de l'espion sera automatiquement modifiée.

En bas à droite, une autre fenêtre permet de voir, entre autres, **la pile des appels**, c'est-à-dire quelle fonction appelle quelle autre fonction. Ici, il n'y en a que deux, donc ce n'est pas étonnant

de voir `main` appeler `fonction`. Mais quand le code se complexifie, il est très utile de voir par quelle fonction on passe et laquelle déclenche le *bug*.

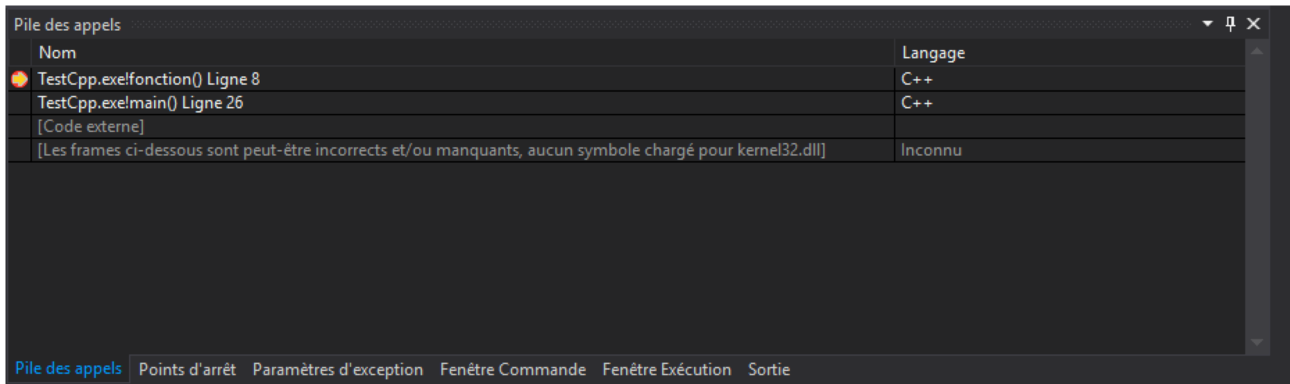


FIGURE 27.5. – Fenêtre de la pile des appels.

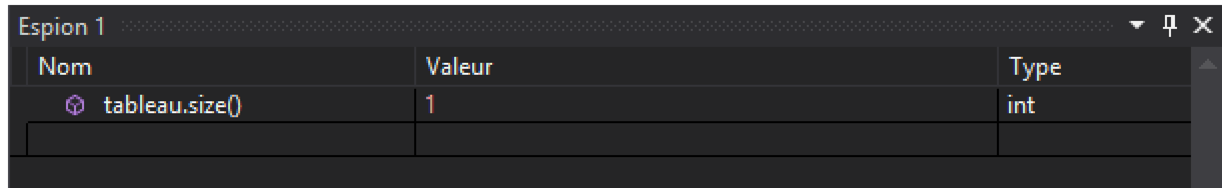
27.2.2. Pas-à-pas

Que diriez-vous maintenant de passer à l'instruction suivante ? Pour cela, on dispose de plusieurs choix possibles, selon ce qu'on décide de faire.

FIGURE 27.6. – Interface avec les différentes actions possibles.

— Le bouton Continuer permet d'exécuter le code **jusqu'au prochain point d'arrêt**. Comme ici il n'y en a qu'un seul, cela signifie jusqu'à la fin du programme.
— Le carré rouge **arrête le débogage**. Avec le clavier, on fait Maj + F5.
— La flèche cerclée **redémarre le débogage**. Avec le clavier, on fait Ctrl + Maj + F5.
— Le **pas-à-pas principal** permet de **passer à l'instruction suivante**, mais sans aller plus loin dans la pile d'appel. Cela signifie que si l'instruction à exécuter est un appel de fonction, le pas-à-pas principal n'ira pas voir dedans. Avec le clavier, c'est F10.
— Le **pas-à-pas détaillé** permet d'examiner **en détails** les instructions. Ainsi, si vous passez sur une fonction, le pas-à-pas détaillé ira dans cette fonction examiner ses instructions. Avec le clavier, c'est F11.

En cliquant sur le pas-à-pas principal, nous voyons la flèche jaune se déplacer d'une ligne et passer à l'instruction suivante. Dans la fenêtre des variables locales, `tableau` s'est mis à jour et sa taille est passée à 1, tout comme l'espion `tableau.size()`. On a donc bien exécuté une instruction `push_back` sur le tableau.

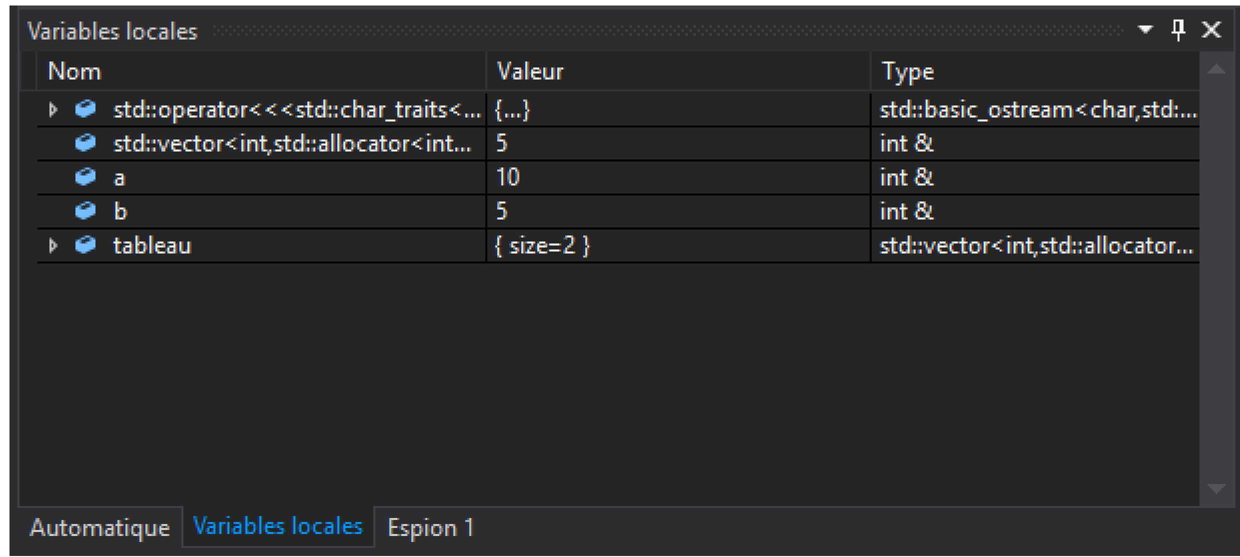


Espion 1

Nom	Valeur	Type
tableau.size()	1	int

Figure 27.7. – Variable espionne bien mise à jour après exécution de la première instruction, qui ajoute un élément au tableau.

L'image ci-dessous vous montre l'état des variables quand je fais un pas-à-pas principal jusqu'à la fin de la fonction. On voit bien que `a` et `b` existent maintenant. On en note également deux autres, dues à la manipulation du flux de sortie, qu'on peut ignorer parce qu'elles ne nous intéressent pas.



Variables locales

Nom	Valeur	Type
std::operator<<<std::char_traits<...	{...}	std::basic_ostream<char,std:...
std::vector<int,std::allocator<int...	5	int &
a	10	int &
b	5	int &
tableau	{ size=2 }	std::vector<int,std::allocator...

Figure 27.8. – Variables après le pas-à-pas principal jusqu'à la fin de la fonction.

27.2.3. Point d'arrêt conditionnel

Faire un pas-à-pas classique, c'est bien, mais quand on doit dérouler une boucle de 100 éléments et qu'on veut la valeur à la 95ème itération, c'est embêtant. Sauf qu'on dispose d'un moyen pratique qui consiste à dire au débogueur de **s'arrêter quand une condition particulière est vérifiée**.

Prenons un code simple comme celui ci-dessus et posons un point d'arrêt sur la ligne 7. En faisant un clic-droit sur le point d'arrêt, puis cliquez sur `Conditions...`. Vous allez voir un cadre apparaitre, dans lequel on peut décider d'une condition à respecter pour que le débogueur s'arrête sur le point d'arrêt désigné.

Figure 27.9. – Point d'arrêt conditionnel.

Si on lance le débogage, on voit bien que le code s'exécute jusqu'au moment où `i` vaut `95`. Là, le programme est stoppé et le déboguer attend nos instructions. Pour preuve, regardez les variables locales.

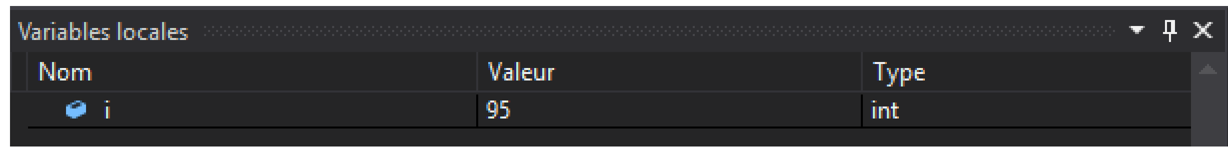


Figure 27.10. – Notre variable locale i vaut bien 95.

27.2.3.1. Aller plus loin

Tout présenter m'est impossible, surtout pour un outil qui mériterait un cours à lui tout seul. Je vous encourage à **jeter un regard sur** les tutoriels de la documentation officielle Microsoft. Apprendre à bien maîtriser le débogueur de Visual Studio ne vous servira pas qu'en C++, mais également en C# ou en Python, si vous continuez à utiliser cet environnement pour développer avec ces langages.

27.3. Qt Creator

Le principe du débogage est assez semblable avec Qt Creator. Cet environnement se base sur l'outil libre gdb , qui s'utilise normalement en ligne de commande.

Si vous lancez le code tel quel en *debug*, la console va s'ouvrir et se refermer aussitôt. Il faut donc commencer par poser des points d'arrêt, ce qui se fait en posant un point d'arrêt en cliquant sur la colonne à gauche de celle des numéros de lignes. On lance ensuite le débogage avec F5.

Voici déjà le code qui servira d'exemple.

27.3.1. Interface

Commençons par examiner l'interface de débogage, pour le même code que ci-dessus. On trouve une petite flèche jaune qui indique **la ligne sur laquelle on se trouve**.

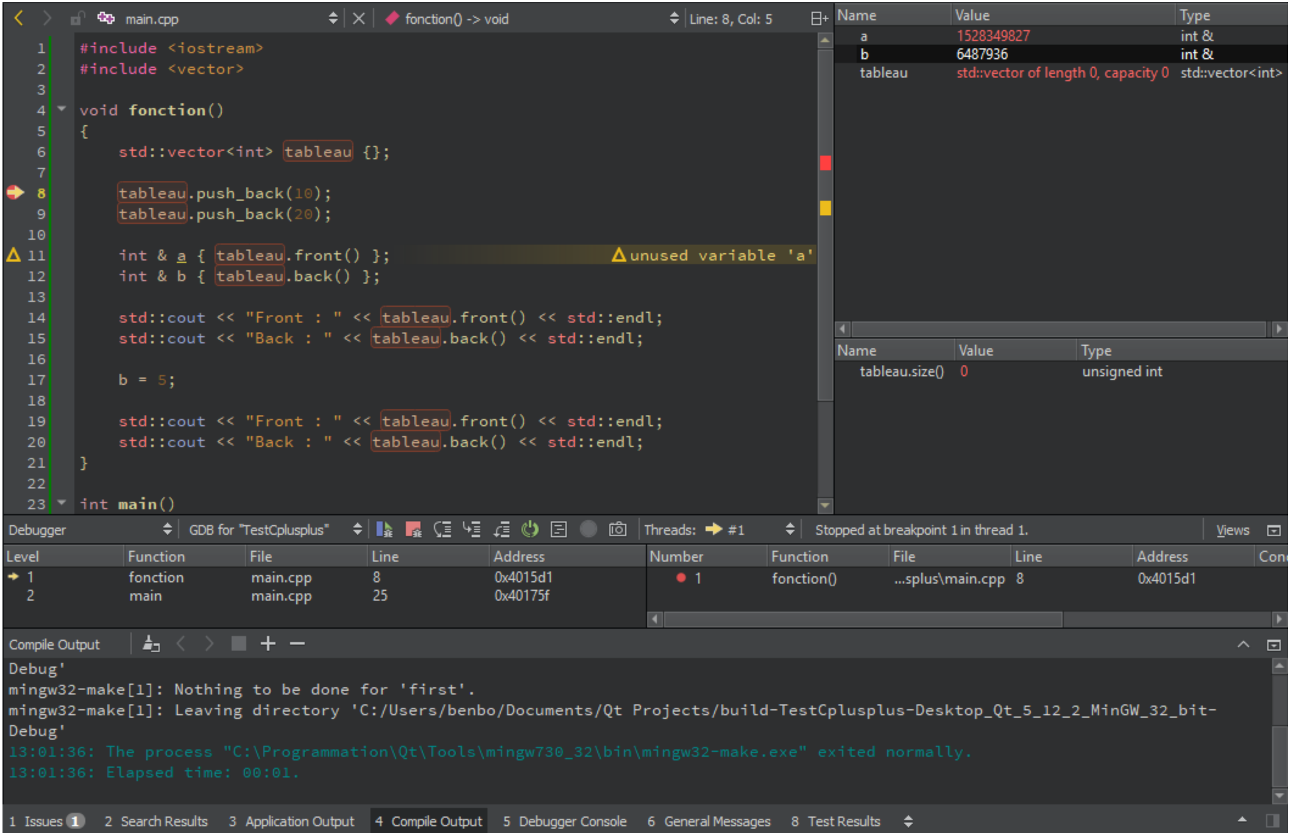


FIGURE 27.11. – Vue de Qt Creator pendant un débogage.

En haut à droite, on a d'abord **la liste des variables** locales et **les paramètres**. On retrouve bien `tableau` qui est vide ; `a` et `b` contiennent des valeurs aléatoires, car elles sont encore non-initialisées.

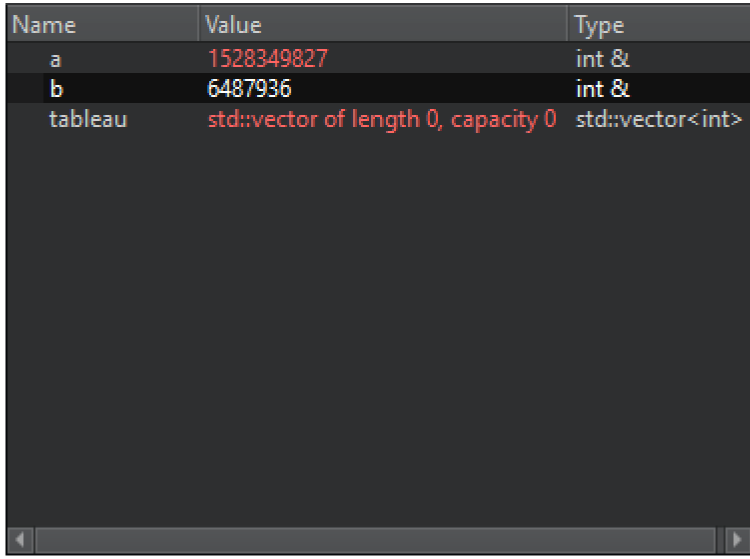


Figure 27.12. – Les variables locales.

Juste en dessous se trouve la fenêtre des **expressions**. On peut ici écrire n'importe quelle expression C++ valide, ce qui permet de toujours garder un œil sur la valeur d'une variable en particulier, comme ici la taille du tableau.

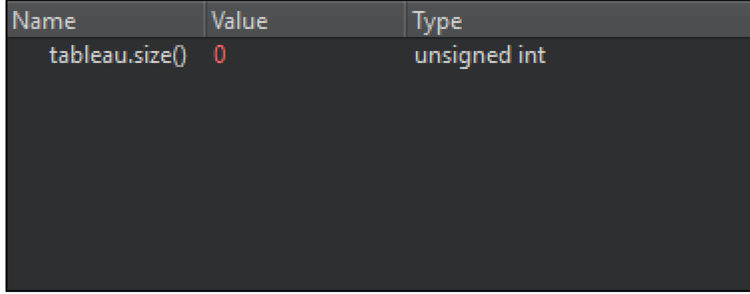


Figure 27.13. – Une expression évaluant la taille du tableau, automatiquement mise à jour quand le tableau est modifié.

Enfin, plus bas, en dessous du code, on trouve **la pile des appels**, c'est-à-dire quelle fonction appelle quelle autre fonction. Également, on trouve la liste de tous les points d'arrêt, ce qui est plutôt utile quand on en a beaucoup.

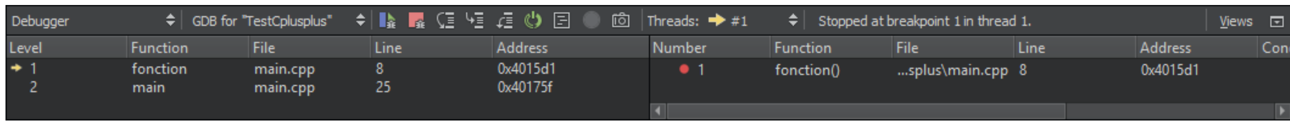


Figure 27.14. – Liste des appels et des points d'arrêt.

27.3.2. Pas-à-pas

Maintenant, il serait de bon ton de continuer le débogage de notre application. Faisons donc du pas-à-pas.

Figure 27.15. – Les différentes options de pas-à-pas.

— Le bouton Continuer permet de **continuer l'exécution jusqu'au prochain point d'arrêt**. Avec le clavier, on fait F5.

— Le carré rouge **arrête le débogage**.

— Le bouton permet le **pas-à-pas principal**, c'est à dire se rendre à la prochaine instruction sans explorer la pile d'appel. Cela signifie que si l'instruction à exécuter est un appel de fonction, le pas-à-pas principal n'ira pas voir dedans. Avec le clavier, c'est F10.

— Le bouton permet le **pas-à-pas détaillé**, c'est à dire se rendre à la prochaine instruction en explorant la pile d'appel. Ainsi, si vous passez sur une fonction, le pas-à-pas détaillé ira dans cette fonction examiner ses instructions. Avec le clavier, c'est `F11`.

— Le bouton **redémarre la session de débogage**.

En cliquant sur le pas-à-pas principal, on voit bien la flèche jaune se déplacer à l'instruction suivante, ajoutant ainsi une valeur au tableau, ce qui met également l'expression évaluée `tableau.size()` à jour.

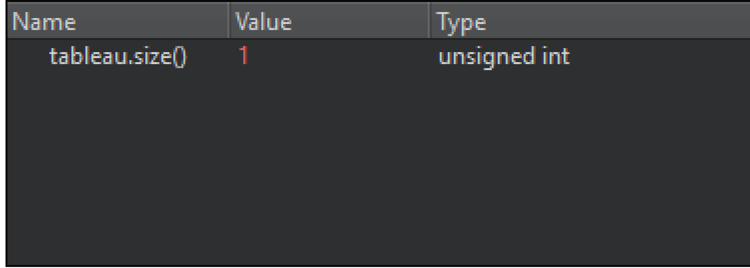


Figure 27.16. – L'expression a bien été mise à jour.

Si on poursuit le pas-à-pas principal jusqu'à la fin de la fonction, on voit que toutes nos variables sont mises à jour et ont maintenant des valeurs cohérentes.

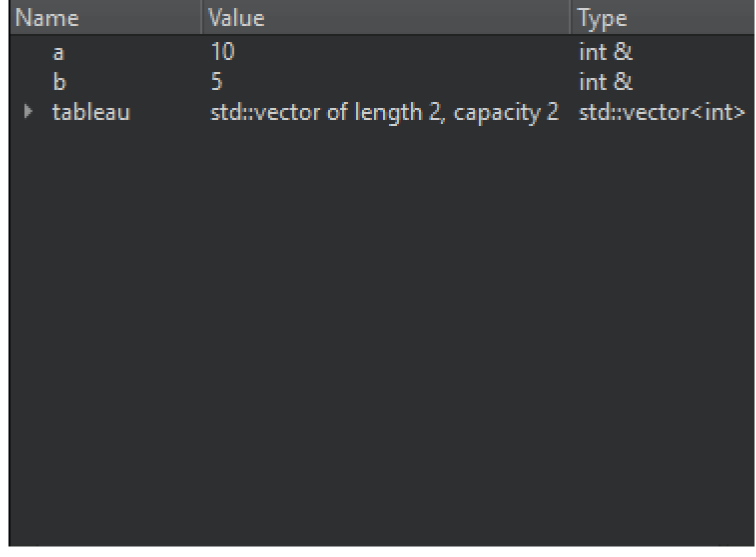


Figure 27.17. – Les valeurs quand le pas-à-pas atteint la fin de la fonction.

27.3.3. Point d'arrêt conditionnel

Avec Qt Creator, on peut tout à fait **poser une condition sur un point d'arrêt**, si l'on veut qu'il s'exécute à une condition précise. Dans le cas d'une boucle, on peut ainsi s'épargner de faire du pas-à-pas et aller directement à une valeur qui nous intéresse. Reprenons le code de la partie précédente et faisons le test.

Pour poser une condition, il suffit de faire un clic-droit sur un point d'arrêt et de choisir Edit breakpoint…. Une fenêtre apparait. Dans la section Advanced, la ligne Condition permet de définir la condition à respecter pour que le débogueur s'arrête sur ce point d'arrêt. On peut ainsi demander à attendre que `i` vaille `95`.

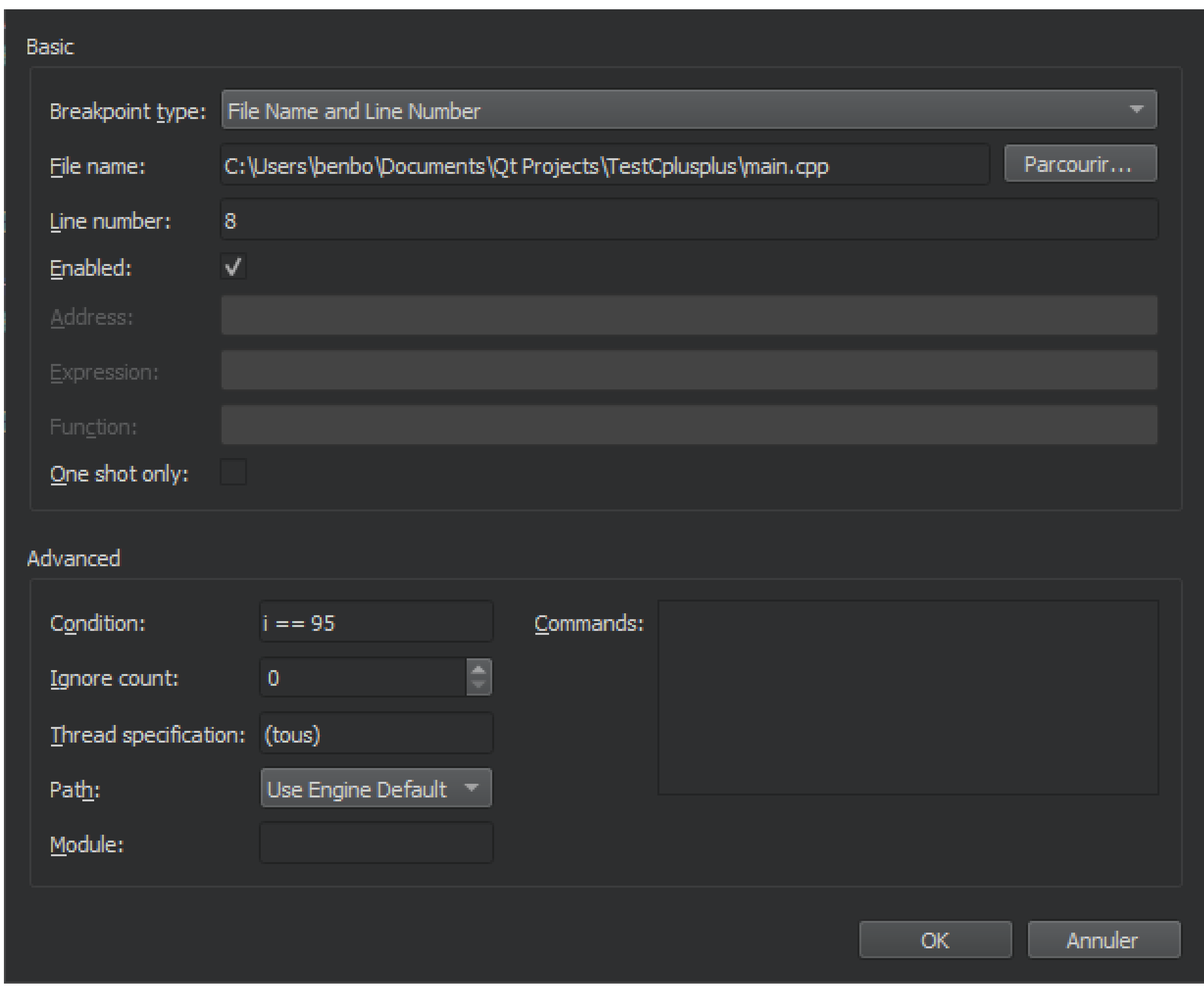


FIGURE 27.18. – Fenêtre de paramétrage d'un point d'arrêt.

Lancez le débogage et vous verrez que le programme s'exécutera sans interruption jusqu'à ce que `i` atteigne la valeur `95`, auquel cas le débogueur attendra nos instructions pour continuer.

27.3.4. Aller plus loin

Comme il m'est impossible d'être exhaustif et que vous commencez à être des habitués de la documentation, je vous renvoie vers **la documentation Qt officielle** . Celle-ci est en anglais mais vous permettra d'aller plus loin dans votre apprentissage.

27.4. En ligne de commande avec gdb

Nous avons dit, dans la partie précédente, que Qt Creator se base sur gdb pour offrir une expérience de débogage. Ce dernier s'utilise avec de nombreuses interfaces graphiques différentes, mais aussi **en ligne de commande**. Alors rouvrons notre shell et plongeons-nous à la découverte de gdb.

Pour lancer le débogage d'un programme, il suffit de taper `gdb` suivi du nom de ce programme. Celui-ci doit avoir été compilé avec les symboles de débogage. L'option `-quiet` évite l'affichage de toutes les informations légales et de licence.

27.4.1. Poser un point d'arrêt

Poser un point d'arrêt avec gdb est faisable de trois façons différentes, selon ce qu'on veut faire.

- On indique un **endroit précis**, comme un numéro de ligne, une fonction, un fichier, etc. C'est un **point d'arrêt** classique (*break point*).
- Lors de la **modification** ou de la **lecture** d'une variable. C'est ce qu'on appelle un **point de surveillance** (*watch point*).
- Lors d'un certain **évènement particulier**, comme le lancement ou le rattrapage d'une exception, entre autres. C'est ce qu'on appelle un **point d'interception** (*catch point*).

Dans notre cas, contentons-nous de poser un point d'arrêt à une ligne précise, la numéro 8 du code suivant, repris des sections précédentes.

Cela se fait en utilisant la commande `break` et en indiquant le numéro de la ligne voulue.

Maintenant, lançons l'exécution du programme avec la commande `run`.

Le débogueur s'est bien arrêté à la ligne 8, comme demandé. Il nous affiche en prime la fonction dans laquelle on se trouve, ainsi que le fichier, puis la ligne de code qui se trouve à la position demandée.

Dans le cas où l'on souhaite poser un point d'arrêt dans un autre fichier, il suffit de préciser celui-ci avant le numéro de la ligne, séparés par deux-point `:`.

27.4.2. Supprimer des points d'arrêt

Quand un point d'arrêt n'est plus utile, on peut **le supprimer**. On a pour cela plusieurs possibilités. Soit on se souvient de la position du point d'arrêt, auquel cas on le supprime avec la commande `clear position`, soit on se souvient de son numéro et on peut le supprimer avec `delete numéro`.

Et comme se souvenir du numéro est compliqué, on peut tout à fait **retrouver la liste des points d'arrêt** avec la commande `info breakpoints`.

27.4.3. Désactiver des points d'arrêt

Parfois, on ne veut pas s'arrêter à un point d'arrêt donné, mais on ne veut pas non plus le supprimer car on en a besoin plus tard. Il suffit de **le désactiver** avec la commande `disable numéro`. Une fois qu'on en aura besoin, il suffira de **le réactiver** avec `enable numéro`.

27.4.4. Afficher l'état d'une variable

Si on regardait l'état de notre tableau ? Celui-ci est bien évidemment vide, mais demandons quand même à `gdb` de nous l'afficher. Il faut pour cela utiliser la commande `print` avec le nom de la variable voulue. Celle-ci doit exister à ce moment-là.

Notre référence `a` existe, mais n'a pas encore de valeur, d'où le nombre aléatoire que vous verrez. Quant à `x`, le symbole n'existe pas car il n'y a aucune variable pourtant ce nom.

27.4.5. Pas-à-pas

Bon, tout ça c'est bien, mais comment on passe à la suite ? Comment exécuter les instructions qui suivent quand on est à un point d'arrêt ? Nous avons deux choix, permettant de faire du **pas-à-pas détaillé**, en explorant la pile des appels, avec `step`, ou bien de **passer d'instruction en instruction** sans explorer le code des fonctions appelées avec `next`.

La sortie suivante montre que, quand je décide de faire `step` sur le point d'arrêt, gdb commence à me montrer en détail ce qui se passe et explore la fonction `std::vector::push_back`. Si je me contente de faire `next`, alors je passe simplement à la ligne d'après, soit le deuxième `push_back`.

Chacune de ces commandes peut recevoir un entier en paramètre, indiquant de combien d'étapes on souhaite avancer. Ainsi, si je suis à la ligne 9 et que je fais `next 3`, j'avancerai de trois instructions. On saute les lignes 10 et 13 qui sont vides, on saute deux instructions aux lignes 11 et 12 et on arrive donc à la troisième après la ligne 9, qui se trouve ligne 14.

27.4.6. Conditions

Dans le cas où l'on voudrait s'arrêter sur un point d'arrêt uniquement lors d'une certaine condition, c'est possible en rajoutant cette condition après `break`. Si je veux, dans le code suivant, m'arrêter quand `i` vaut `95`, je n'ai qu'à taper la commande ci-après.

27.4.7. Aller plus loin

Ce programme est vraiment complet et doté de nombreuses possibilités. Je vous encourage à en apprendre plus de votre côté, que vous vouliez l'utiliser en ligne de commande ou à travers l'interface graphique de votre choix. Vous pouvez commencer par ce tutoriel en français, qui vous montrera d'autres commandes possibles.

27.4.8. En résumé

- Le débogueur est un précieux allié qui peut nous aider à traquer les *bugs* vicieux cachés dans nos codes.
- En posant des points d'arrêt, on demande au débogueur d'arrêter l'exécution du programme à une ligne précise.
- Certains points d'arrêt ne s'activeront que si une condition que nous avons définie devient vraie.
- On peut afficher la pile d'appel, pour voir la liste des fonctions appelées avant d'arriver à une ligne quelconque.
- On peut voir, à tout moment, l'état des variables et même définir nos propres expressions, ou espions, qui seront automatiquement mis à jour à la moindre modification.
- Il ne faut pas oublier de compiler le programmer avec ses symboles de débogage.

28. Une foule de bibliothèques

La bibliothèque standard, vous commencez à connaître et à vous sentir à l'aise avec. Mais soyons honnêtes, elle est limitée. Par exemple, impossible de faire communiquer deux programmes par le réseau ou de faire un jeu 2D avec. Ce n'est pas un problème, **il suffit d'installer des bibliothèques externes**, déjà programmées et testées par d'autres personnes.

Ce chapitre va aborder **l'installation et l'utilisation de bibliothèques externes**.

28.1. Quelles bibliothèques choisir ?

La réponse à cette question dépend de ce que vous cherchez. C++ étant un langage populaire, vous trouverez des bibliothèques externes dans de nombreux domaines, comme les graphismes 2D, la communication réseau, la manipulation de sons, des outils mathématiques supplémentaires, etc. Ceci étant dit, nous sommes dans un cours d'introduction, donc je me dois de choisir pour vous. Dans ce chapitre, nous apprendrons donc à **installer et utiliser brièvement deux bibliothèques externes**, respectivement nommées **Boost** et **SFML**.

Vous aurez ainsi vu plusieurs manières d'installer une bibliothèque externe, donc vous ne serez pas perdus lorsque vous en aurez besoin.

Pas d'inquiétude

Une fois le principe bien compris, installer et utiliser d'autres bibliothèques externes n'aura rien de compliqué, il suffira de lire la documentation.

28.1.1. Boost

La première que nous verrons, Boost, c'est un peu comme **l'extension officieuse de la bibliothèque standard C++**. Au fil des années, de nombreuses fonctionnalités implémentées dans Boost se sont retrouvées intégrées à la bibliothèque standard C++. Citons par exemple `std::array`, `std::tuple` ou encore de nombreuses fonctions mathématiques.

L'installer, c'est s'offrir les services d'une bibliothèque de **grande qualité**, **répandue**, conçue par des développeurs experts, donc vous avez tout à gagner à apprendre à l'utiliser. Elle propose, entre autres, des **tests unitaires**, mais également un moyen d'interfacer son code avec Python, travailler avec des quaternions et bien d'autres choses encore.

28.1.2. SFML

L'autre que j'aimerai que vous découvriez est une **bibliothèque 2D** très connue et utilisée en C++, SFML . Elle vous permet de gérer **tout ce qui touche aux jeux 2D**, comme le son, les graphismes et le réseau. Et ne vous inquiétez pas, **elle est portable** et fonctionne sur de nombreuses plateformes, dont Windows et GNU/Linux.

FIGURE 28.1. – Exemple utilisant la SFML, tiré de la documentation.

En plus, la documentation est en français. C'est pas beau la vie ?

28.2. Généralités

La plupart des bibliothèques externes fonctionnent sur le même principe que la bibliothèque standard, c'est-à-dire qu'on a des **fichiers d'en-têtes** et des **fichiers objets**, entre autres. Les premiers sont nécessaires pour la compilation, les seconds pour le *linker*, comme nous l'avons vu dans le chapitre sur la compilation. Mais ce n'est pas tout. Souvent sont joints des **exemples**, ainsi qu'une version hors-ligne de la **documentation**.

28.2.1. Statique ou dynamique ?

Les fichiers objets d'une bibliothèque externe peuvent être joints au programme de deux manières différentes.

— Avec une **bibliothèque statique**, les fichiers objets sont **directement liés au programme**. Celui-ci devient plus gros, mais est désormais utilisable tel quel, sans fichier supplémentaire à joindre avec l'exécutable.

— Avec une **bibliothèque dynamique**, les fichiers objets sont chargés **à l'exécution** du programme. L'avantage, c'est qu'une même copie de la bibliothèque peut être utilisée par de nombreux programmes. L'inconvénient, c'est qu'il faut fournir ces fichiers avec l'exécutable du programme. Sous Windows, ce sont les fameux fichiers `.dll`.

La très grande majorité des bibliothèques externes sont utilisables de façon statique ou dynamique, au choix.

28.2.2. *Debug* ou *release* ?

De la même manière que nos programmes, les bibliothèques externes sont **compilées tant en *debug* qu'en *release***. Cela permet d'offrir une expérience de débogage et des tests pendant qu'on développe notre programme, avec les symboles de débogage présents et une expérience optimisée et plus rapide une fois qu'on passe en *release*.

28.3. Installer Boost

Commençons donc par Boost. Cette bibliothèque est composée majoritairement de fichiers d'en-tête, qu'on peut inclure simplement, sans plus d'histoire. Cependant, quelques parties demandent **d'être compilées**. Nous allons donc voir comment installer la version complète de Boost sur chaque plateforme.

28.3.1. GNU/Linux

Vous êtes dans le cas le plus facile, puisqu'il y a de grandes chances que votre distribution préférée fournisse des paquets prêts à l'emploi. Par exemple, pour Ubuntu, c'est la commande suivante qui installe le tout.

```
```

Renseignez-vous dans la documentation de votre distribution pour le nom exact du paquet.

28.3.2. Windows

Les utilisateurs de Windows sont moins chanceux. De base, il faudrait **compiler nous-mêmes une version de Boost**. Pour ceux utilisant Visual Studio, une solution simple existe. Pour ceux utilisant Qt Creator…

28.3.2.1. Visual Studio

Commencez par vous rendre ici . Sur la page des versions, prenez la dernière disponible (1.69 à l'heure où j'écris), puis téléchargez **boost_1_69_0-msvc-14.1-32.exe**, ainsi que **boost_1_69_0-msvc-14.1-64.exe** si vous avez un système d'exploitation 64 *bits*. Une fois l'exécutable téléchargé, installez-le dans le dossier de votre choix. Si vous prenez les deux, installez les deux dans le même dossier. Ne vous en faites pas, il n'y a pas de risque d'écrasement.

Maintenant, ouvrez un projet quelconque et rendez-vous dans ses propriétés, avec un clic-droit sur le nom du projet. Il va en effet falloir indiquer au compilateur **où se trouvent les fichiers de Boost**. Commençons par nous mettre en configuration générale, en mettant le champ Configuration à Toutes les configurations et Plateforme à Toutes les plateformes.

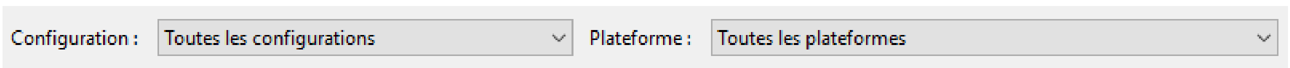


FIGURE 28.2. – Ce que vous devez avoir.

Incluons le dossier contenant tous les fichiers d'en-tête en allant dans C/C + -> Général, puis en faisant modifier sur le champ Autres répertoires Include. Dans le cadre du haut, mettez le chemin du dossier dans lequel vous avez installé Boost. Chez moi, le chemin complet est **C :\Programmation\Boost.1.69**.

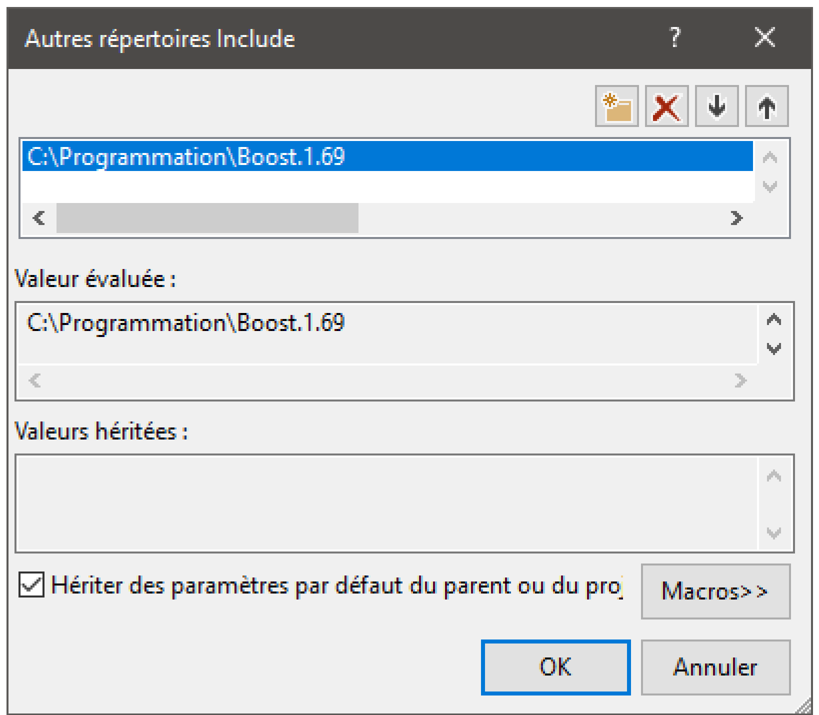


FIGURE 28.3. – Ce que vous devez avoir.

Maintenant, allons dans Éditeur de liens -> Général et modifions le champ Répertoires de bibliothèques supplémentaires pour rajouter le dossier contenant les fichiers objets. Dans le cas où vous n'avez que la version 32

bits, modifiez-le et ajoutez le chemin où vous avez installé Boost plus **lib32-msvc-14.1**. Chez moi, le chemin complet est **C :\Programmation\Boost.1.69\lib32-msvc-14.1**.

Dans le cas où vous avez les deux versions, 32 *bits* et 64 *bits*, il va falloir **adapter le chemin donné**. Tout d'abord, modifiez le champ Plateforme en haut pour le mettre à Win32. Puis, dans Éditeur de liens -> Général, modifiez le champ Répertoires de bibliothèques supplémentaires. Ajoutez-y le chemin où vous avez installé Boost plus **lib32-msvc-14.1**. Une fois fait, cliquez sur Appliquer puis retourner modifier le champ Plateforme en le mettant cette fois à x64. Faîtes la même chose que précédemment mais en sélectionnant cette fois le dossier **lib64-msvc-14.1**

— Pour Win32, j'ai dans mon cas le chemin **C :\Programmation\Boost.1.69\lib32-msvc-14.1**.
— Pour x64, j'ai dans mon cas le chemin **C :\Programmation\Boost.1.69\lib64-msvc-14.1**.

Validez et tout est bon.

28.3.2.2. Qt Creator

Quant à ceux utilisant Qt Creator, ils gagnent le droit de **compiler eux-mêmes Boost**. Ne vous inquiétez pas, ce n'est pas si compliqué qu'il n'y parait et je vais tout vous expliquer. Il faut commencer par télécharger la dernière version disponible (1.69 à l'heure où j'écris) et la décompressez dans le dossier de votre choix.

Maintenant, on va lancer l'invite de commande installé en même temps que Qt Creator. Dans mon cas, je le trouve dans Menu démarrer -> Qt -> 5.12.2 -> MinGW 7.3.0 (32 *bits*) -> Qt 5.12.2 (MinGW 7.3.0 32-bit) (les numéros de version peuvent changer). Il va nous servir à compiler Boost.

Rendez-vous dans le dossier où vous avez décompressé Boost à l'aide de la console. Dans mon cas, c'est **C :\Users\informaticienzero\Downloads\boost_1_69_0**. La commande pour changer de dossier est **cd**.

Une fois dedans, on va préparer l'installation en exécutant la commande suivante. Entre guillemets après `--prefix`, mettez le chemin de sortie où les bibliothèques composant Boost seront stockées. Pour moi, j'ai pris **C :\Programmation\Boost**.

Maintenant, il est temps de lancer la compilation pour de bon. Voici la ligne à taper. Oui, elle est très longue.

— L'option `address-model=32` indique qu'on veut compiler une version 32 *bits* de la bibliothèque.

— L'option `architecture=x86` indique qu'on compile pour un PC, qui utilisent quasiment tous cette architecture.
— L'option `--build-dir="C:\Programmation\Boost"` indique où on veut stocker les fichiers objets générés.
— Les options `debug release` indique qu'on veut disposer tant de la version *debug* que *release*.
— L'option `link=static` indique qu'on veut compiler une version statique uniquement, ce qui est fortement recommandé sous Windows.
— L'option `cxxflags="-std=c++17"` indique qu'on veut compiler avec C++17.

La compilation peut être assez longue donc lancez-la et faites autre chose en attendant. Une fois qu'elle a fini, il faut configurer Qt Creator. Cela se passe dans le fichier **.pro**. On commence par indiquer où trouver les fichiers d'en-tête.

Maintenant, on va ajouter le dossier contenant les **fichiers objets**.

Voilà, l'installation est terminée, il ne restera plus qu'à ajouter individuellement les bibliothèques requises dans le fichier **.pro**.

> *i* Version 64 bits
>
> Si vous voulez la version 64 *bits* de Boost, il faut refaire la même procédure mais avec la console **Qt 5.12.2 (MinGW 7.3.0 64-bit)**.

28.3.3. Tester l'installation

Prenons par exemple le code d'illustration de Boost.Log , un système d'écriture de messages de journalisation. Je le remets ci-dessous.

Pour compiler en ligne de commande, il faut rajouter quelques options, dues à des dépendances du système de *logging* fourni par Boost.

— L'option `-static` indique au compilateur qu'on veut lier **statiquement** Boost.Log à notre programme.
— Les options `-lboost_log` et `-lboost_log_setup` permettent de lier les deux bibliothèques nécessaires à Boost.Log.
— Les options `-lboost_thread`, `-lboost_system` et `-lpthread` lient d'autres dépendances de Boost.Log à notre programme. On trouve notamment Boost.Thread et Boost.System, deux autres bibliothèques inclues par Boost, ainsi qu'une dépendance à une bibliothèque commune à toutes les distributions GNU/Linux et UNIX.

Pour ceux utilisant Qt Creator, on doit ajouter les lignes suivantes dans le fichier **.pro**. Ce sont les bibliothèques nécessaires pour compiler le code d'exemple de Boost.Log. On retrouve `log` et `log_setup`, ainsi que la bibliothèque `thread`. Notez qu'on fait la différence entre les versions *debug* (avec **d**) et *release* (sans **d**).

> **! 32 bits et 64 bits**
>
> Si vous compilez avec MinGW 32 *bits*, alors il faut lier les bibliothèques 32 *bits* ; pareillement pour 64 *bits*.

Dans tous les cas, une fois compilé et lancé, le programme affichera la sortie suivante, avec seulement l'heure changeant en fonction du moment où vous lancerez le programme.

28.4. Installer SFML

Maintenant, abordons l'installation de la deuxième bibliothèque, la SFML. Encore une fois, nous allons détailler par environnement.

28.4.1. GNU/Linux

Vous êtes encore des chanceux, puisque la plupart des distributions proposent des paquets permettant d'avoir une SFML à jour en une commande. Ainsi, avec Ubuntu, il suffit de taper la commande qui suit.

Renseignez-vous pour le nom exact du paquet pour votre distribution.

28.4.2. Windows

Pour Windows, il va falloir installer manuellement. Pas de soucis, la documentation est très bien faite, en français et vous bénéficiez de nos explications en prime. Commençons par nous rendre sur la page de téléchargement . À l'heure où j'écris, elle propose la version 2.5.1 de SFML.

Plusieurs versions sont disponibles au téléchargement, en fonction du compilateur que vous utilisez. Dans notre cas, il s'agira de prendre soit la version **Visual C++ 15 (2017)** pour ceux ayant installé Visual Studio 2017, soit la version **GCC 7.3.0 MinGW** pour ceux utilisant Qt Creator. Dans les deux cas, prenez la version **32 *bits***. Même si la très grande majorité des processeurs vendus aujourd'hui sont 64 *bits*, avec la version 32 *bits* vous pourrez cibler les deux-plateformes. Si vous prenez la version 64 *bits*, vous ne pourrez cibler que les ordinateurs ayant un processeur et un système d'exploitation 64 *bits*.

Dans tous les cas, vous obtenez une archive que vous pouvez décompresser **dans le dossier de votre choix**. Certains vont le mettre dans un dossier bien à part, comme `C:/Programmation/SFML`, d'autres préfèrent le mettre dans le dossier de leur projet , ce qui permet de distribuer les sources et de changer d'ordinateur en un simple copier-coller, car la configuration est enregistrée avec le projet. Ce choix est vôtre.

28.4.2.1. Avec Visual Studio

Voici tout d'abord la documentation détaillant l'installation avec Visual Studio. Elle est très claire et vous pouvez la suivre sans mon aide, vous en avez largement les capacités.

Néanmoins, pour ceux qui se sentiraient rassurés en me voyant faire, lisez la suite. D'abord, il faut modifier les propriétés de configuration du projet, en faisant un clic-droit dessus. Tout en haut à gauche, assurez-vous d'être en mode Toutes les configurations ainsi que Toutes les plateformes.

Figure 28.4. – Voici ce que vous devez avoir.

Il y a deux choses à faire qui sont communes à *debug* et *release*.

- Dans C/C+ -> Général, dans le champ Autres répertoires Include, faites modifier, puis dans le cadre du haut mettez le chemin du dossier **include** du répertoire où vous avez installé SFML.
- Dans Éditeurs de liens -> Général, dans le champ Répertoires de bibliothèques supplémentaires, faites modifier, puis dans le cadre du haut mettez le chemin du dossier **lib** du répertoire où vous avez installé SFML.

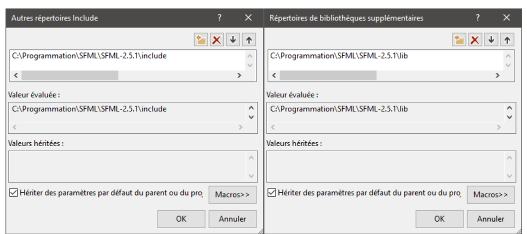


Figure 28.5. – Voilà ce que je mets dans mon cas.

Comme SFML est divisée en plusieurs modules, il faut ajouter plusieurs bibliothèques en particulier pour que notre code fonctionne. Dans l'étape précédente, on a indiqué au *linker* où chercher. Maintenant, on va indiquer exactement celles qu'on veut qu'il prenne.

Cette fois, le travail est à faire en deux fois, pour *debug* et pour *release*. Passez donc en configuration Debug puis allez dans Éditeur de liens -> Entrée. Là, modifiez le champ Dépendances supplémentaires, faites modifier puis ajouter, dans le cadre du haut, les valeurs suivantes.

- sfml-graphics-d.lib
- sfml-system-d.lib
- sfml-window-d.lib

Nom de la bibliothèque

Le **d** dans le nom des fichiers indique qu'on a affaire à la version *debug* de la bibliothèque. Cette convention est souvent reprise.

Faite la même chose en mode Release mais prenez cette fois les versions ne contenant pas **d**.

- sfml-graphics.lib
- sfml-system.lib
- sfml-window.lib

La configuration du projet est terminée, mais il reste une dernière étape. SFML ayant été installée de manière dynamique, il faut ajouter les DLLs, contenues dans le dossier **bin**, au projet. Copiez-collez les dans le répertoire du projet et tout est bon.

28.4.2.2. Avec Qt Creator

Ouvrez votre projet, faites un clic-droit dessus et cliquez sur Ajouter une bibliothèque…. Dans la fenêtre suivante, cliquez sur **Bibliothèque externe**. Sur la fenêtre suivante, cliquez sur **Fichier de bibliothèque** puis rendez-vous dans le dossier **lib** de la SFML et choisissez **libsfml-main.a**. Dans la liste des plateformes, décochez Linux et Mac. Enfin, vérifiez que l'option **Ajouter un suffixe "d" pour la version debug** soit cochée.

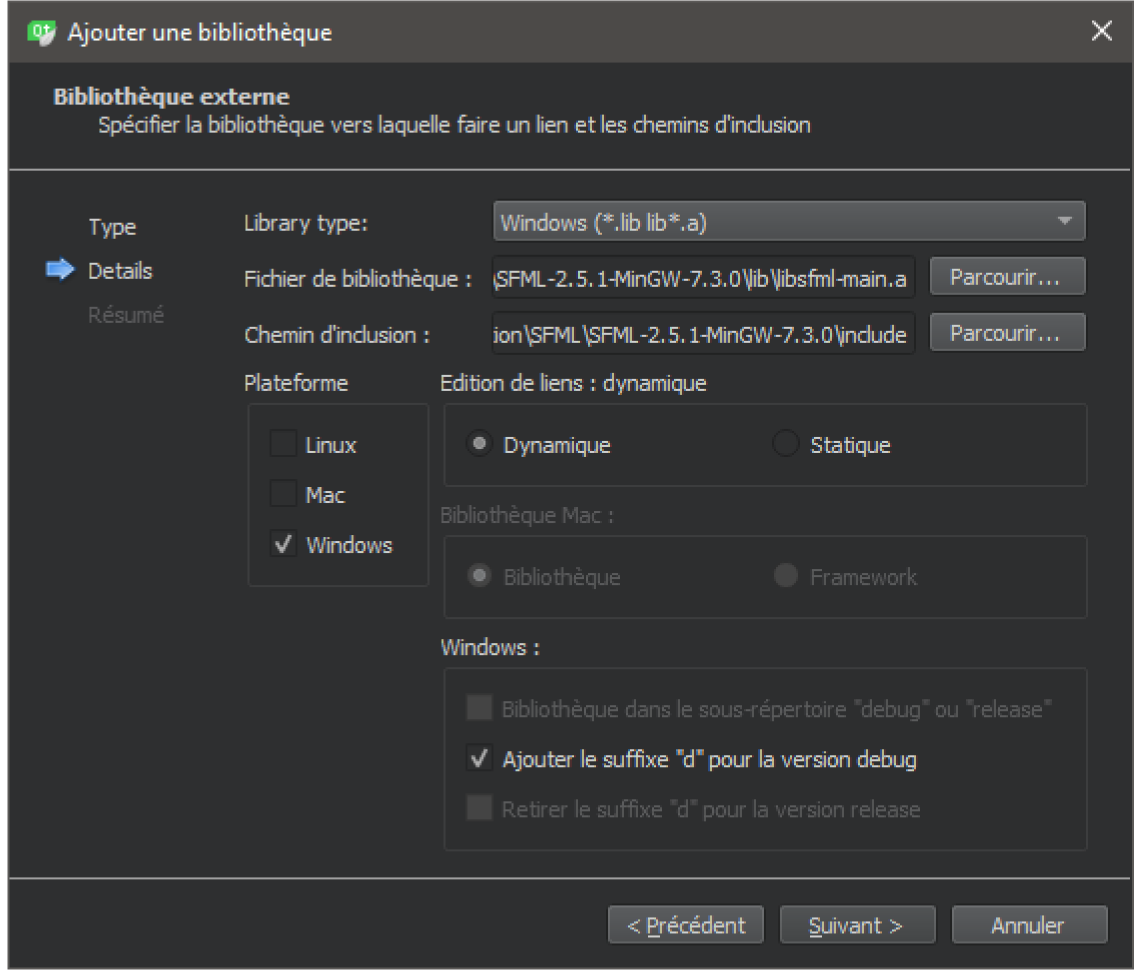


FIGURE 28.6. – Voilà ce que vous devriez avoir.

Validez et ouvrez votre fichier **.pro**, que nous allons devoir modifier. De nouvelles lignes ont en effet été ajoutées.

Ces lignes indiquent où chercher les fichiers d'en-tête que nous incluons, nous n'allons donc plus y toucher. Par contre, les deux autres lignes nécessitent une petite modification.

La version de *debug* n'est pas bonne, car il manque un tiret `-`. Il faut donc remplacer `-lsfml-maind` par **`-lsfml-main-d`**. Ensuite, comme nous voulons charger d'autres modules, nous allons dupliquer ces lignes plusieurs fois, pour être en mesure d'ajouter **libsfml-window**, **libsfml-graphics** et **libsfml-system**. Voici les lignes à ajouter.

Enfin, il reste à **ajouter les DLLs au dossier de sortie**. Toutes celles contenant `-d` vont dans le répertoire *Debug*, celles sans dans le répertoire *Release*.

28.4.3. Tester l'installation

Il ne reste plus qu'à tester en utilisant le code donné en exemple à la fin de la page de documentation. Si vous voyez un cercle vert s'afficher, c'est gagné.

FIGURE 28.7. – Un beau cercle vert en guise de résultat.

28.4.4. En résumé

- Quand une fonctionnalité dont nous avons besoin n'existe pas dans la bibliothèque standard, on peut choisir d'installer des bibliothèques externes.
- Boost est une des bibliothèques externes les plus célèbres et les plus utilisées en C++. Elle comporte de nombreuses fonctionnalités comme le *logging*, des fonctions mathématiques ou la communication réseau.
- SFML est une bibliothèque réputée permettant de faire, entre autres, des jeux 2D.
- Chaque bibliothèque externe vient avec sa documentation expliquant comment l'installer et l'utiliser. C'est votre meilleure amie et le premier réflexe à avoir quand vous avez besoin d'une information.

29. Améliorer ses projets

Vous avez, au cours de cette troisième partie, énormément appris sur ce qu'être un développeur signifie. Bien entendu, il y a encore de nombreuses choses à apprendre et cette section ne pourra jamais être exhaustive. Il est temps pour nous de la conclure avec un dernier chapitre présentant quelques outils qui vous serons quasi-indispensables pour passer au niveau supérieur.

Ce chapitre va présenter plusieurs outils pour **améliorer le développement de vos projets**.

29.1. git — Sauvegarder et versionner son code

29.1.1. La problématique

Sauvegarder, c'est important. Peu importe qu'on parle de code, de photos, de documents, ne pas avoir de sauvegarde peut nous créer de gros problèmes quand un document est corrompu ou effacé par erreur. Ça nous est tous arrivé au moins une fois et c'est très embêtant.

Nos codes n'échappent pas à cette logique. Il est important de **sauvegarder notre code régulièrement**. Quand nos programmes sont tout petits, une solution consistant à copier/coller le dossier du projet peut suffire. Mais cette solution apporte en réalité plus d'inconvénients qu'autre chose.

- Nous n'avons aucun moyen de voir **l'historique des modifications** d'un fichier.
- C'est très **lourd** et **encombrant** puisqu'on se retrouve avec de plus en plus de dossiers.
- Si deux dossiers ne contiennent pas les mêmes fichiers, **les fusionner peut s'avérer complexe**.
- Dans le cas où on travaille à plusieurs sur le projet, **impossible de savoir qui a quelle version**.
- En cas de *bug*, **revenir à une ancienne version est complexe**, voire impossible.

Heureusement et comme toujours, des gens très doués ont créé des outils adaptés, efficaces et il ne nous reste qu'à les utiliser. Ces outils s'appellent des **gestionnaires de code source** . Ils répondent à toutes les problématiques que nous avons soulevées plus haut. Le plus célèbre, utilisé et apprécié s'appelle **git** .

29.1.2. Installation

Dans le cas où vous êtes sous Windows, téléchargez la version la plus récente et installez-la dans le répertoire de votre choix. Pour ceux utilisant GNU/Linux, le paquet `git` est très certainement déjà installé. Dans le cas contraire, ce qui serait extrêmement étonnant, vous pouvez l'obtenir avec votre gestionnaire de paquet.

Bien qu'il existe des surcouches graphiques, je vous conseille d'apprendre à **utiliser git en ligne de commande**. De cette manière, vous serrez autonomes peu importe le système d'exploitation ou l'environnement sur lesquels vous travaillerez, et vous en aurez une utilisation **plus souple**. Nous allons voir les bases dans ce cours, mais pour aller plus loin, il faudra vous baser sur des ressources externes. Je vous conseille notamment celle-ci , pour un guide de base, et le Pro Git Book , une référence très complète, et traduite en français qui plus est.

> **Outil fondamental**
>
> Git est vraiment un outil fondamental pour le développeur, il est très rentable de passer du temps à apprendre à le maîtriser. À la fin de ce cours, je vous suggère fortement de jeter un œil attentif au Pro Git Book.

Sous GNU/Linux, vous n'avez qu'à lancer les commandes directement depuis le terminal. Pour Windows, il faut lancer **Git Bash**, qui se trouve dans le dossier **Git** du Menu Démarrer.

29.1.3. Initialiser git

La première chose à faire est d'**entrer un identifiant**, une **adresse mail** et **l'éditeur de votre choix**. Cela permet de tracer qui a fait quelle modification. Les deux premières commandes à entrer sont les suivantes.

29.1.4. Créer un dépôt

Maintenant, nous allons indiquer à git que nous voulons **sauvegarder le dossier contenant notre code source**. Il suffit de se positionner dans le répertoire voulu et de lancer la commande `git init`. Pour Windows, je procède ainsi, avec un projet Visual Studio en exemple.

Pour GNU/Linux, faites de même. Je prends ici un simple répertoire qui contient mes sources.

Votre projet est maintenant géré par git. C'est aussi simple que ça.

29.1.5. Connaître l'état de ses fichiers

Et si nous demandions à git de **nous donner les fichiers modifiés** ? Cela se fait très simplement en tapant `git status`. Pour le cas d'un projet fraichement initialisé, aucun fichier n'étant suivi, on obtient un message proche du message suivant.

Modifions donc un fichier quelconque. Dans mon cas, je supprime tout le contenu de `main.cpp` pour ne plus afficher qu'un message. Je relance la commande précédente et cette fois, il y a du changement.

On voit bien que git a noté la modification du fichier `main.cpp`. Et si j'ajoute un nouveau fichier, comme un simple document texte ?

Cette fois, il nous indique qu'un fichier non suivi a été ajouté. Cela signifie que git ne le versionnera pas et n'enregistrera pas son historique.

29.1.6. Ajouter un fichier

Il faut donc être en mesure de dire à git de **suivre, ou de ne plus suivre, tel ou tel fichier**. Dans le premier cas, cela se fait avec la commande `git add nom_fichier`.

Le fichier ajouté se trouve maintenant sous la section « *Changes to be committed* », soit les changements à enregistrer. Au contraire, le fichier `main.cpp` ne le sera pas. Il faut l'ajouter lui aussi, pour que les modifications apportées à ce fichier soit elles aussi enregistrées.

29.1.7. Valider les modifications

Nous avons vérifié, tout est bon, nous voulons enregistrer ce que nous avons fait. On parle de **commiter** ou de **faire un *commit***. On utilise la commande suivante `git commit -m "Message"`, qui permet d'enregistrer les modifications avec un message explicatif.

Message explicatif

Le message accompagnant le commit doit être **de qualité**, puisqu'il indique ce que vous avez fait. Des messages comme `correction`, `xxx` ou `changement` sont donc très mauvais puisqu'ils ne sont pas clairs du tout.

29.1.8. Voir l'historique

Vous pouvez voir l'évolution de votre code en demandant à git de vous afficher l'historique des *commits*. Dans notre cas, nous en avons trois. Le premier affiché est le dernier en date, celui que nous venons de faire. Les deux autres ont automatiquement été faits par git lors de l'initialisation du dépôt.

Notez que l'auteur et la date sont affichés, ainsi que le *checksum*, la somme de contrôle identifiant de manière unique chaque *commit*.

29.1.9. Bloquer certains fichiers

Si versionner le code est une bonne pratique, certains fichiers ne doivent pas être intégrés au dépôt. C'est le cas des fichiers objets, entre autres. Ceux-ci sont des produits de la compilation, donc rapidement périmés, pas adaptés en cas de passage à un autre système d'exploitation ou un autre compilateur, etc. On veut donc **les exclure**. De même, on n'inclut généralement pas les exécutables, les fichiers de configuration de l'éditeur utilisé, etc.

Pour ce faire, on dispose d'un fichier **.gitignore**, un bête fichier texte qui indique à git que certains fichiers ou certaines extensions sont à exclure. Il suffit de le créer et d'ajouter dedans **les règles souhaitées**. Par exemple, exclure tous les fichiers objets avec l'extension `.o` se fait en ajoutant la ligne `*.o`.

Astuce Windows

Si vous tentez de créer un fichier nommé **.gitignore** avec l'explorateur Windows, celui-ci protestera que vous n'avez pas entré de nom de fichier. Une astuce consiste à créer un fichier s'appelant **.gitignore.** (notez le point final supplémentaire), que Windows va automatiquement renommer correctement.

Dans le cas d'un projet Visual Studio, je me mets à la racine du projet, dans le même dossier où se trouve la solution **.sln** et je crée le fichier **.gitignore**. Les valeurs que je mets dedans sont celles que Visual Studio recommande, que j'ai pris de précédents projets et que je vous mets

ci-dessous. On retrouve notamment les dossiers *Debug* et *Release* qui sont exclus, ainsi que les fichiers objets `.obj`.

Fichier .gitignore pour Visual Studio

Pour GNU/Linux, pareil, créer ce fichier puis ajouter dedans les lignes ci-dessous.

Fichier .gitignore pour GNU/Linux

Je vous fournis ici des fichiers pré-remplis, mais quand vous aurez acquis de l'expérience, vous saurez les adapter à vos projets et vos contraintes.

29.1.9.1. Exemple concret

Je suis sous GNU/Linux et je compile un fichier source pour obtenir mon programme. Sans fichier `.gitignore`, mon exécutable est inclus aussi et je ne veux pas.

Si je crée mon fichier `.gitignore` comme vu juste avant, cette fois, mon exécutable est bien exclu et je peux commiter mon code.

29.1.10. Aller plus loin

Comment faire pour supprimer un fichier ? Comment revenir à un commit précédent ? Qu'est-ce que le concept de branches et comment l'utiliser ? Toutes ces questions trouvent leurs réponses dans le Pro Git Book, que je vous encourage une fois de plus à lire. Il est d'excellente qualité, progressif et vous enseignera tout ce que vous devez savoir sur git.

29.1.11. Visual Studio

Il est possible de gérer son code source avec git **directement depuis Visual Studio**. Regardez si vous avez la même petite ligne en bas à droite.

Ajouter au contrôle de code source

FIGURE 29.1. – Avez-vous cette ligne ?

Si ce n'est pas le cas, il suffit d'installer une extension. Dans le menu Démarrer, cherchez le programme qui s'appelle **Visual Studio Installer**. Ce programme accompagne Visual Studio 2017 et permet d'installer des mises à jour et des composants supplémentaires.

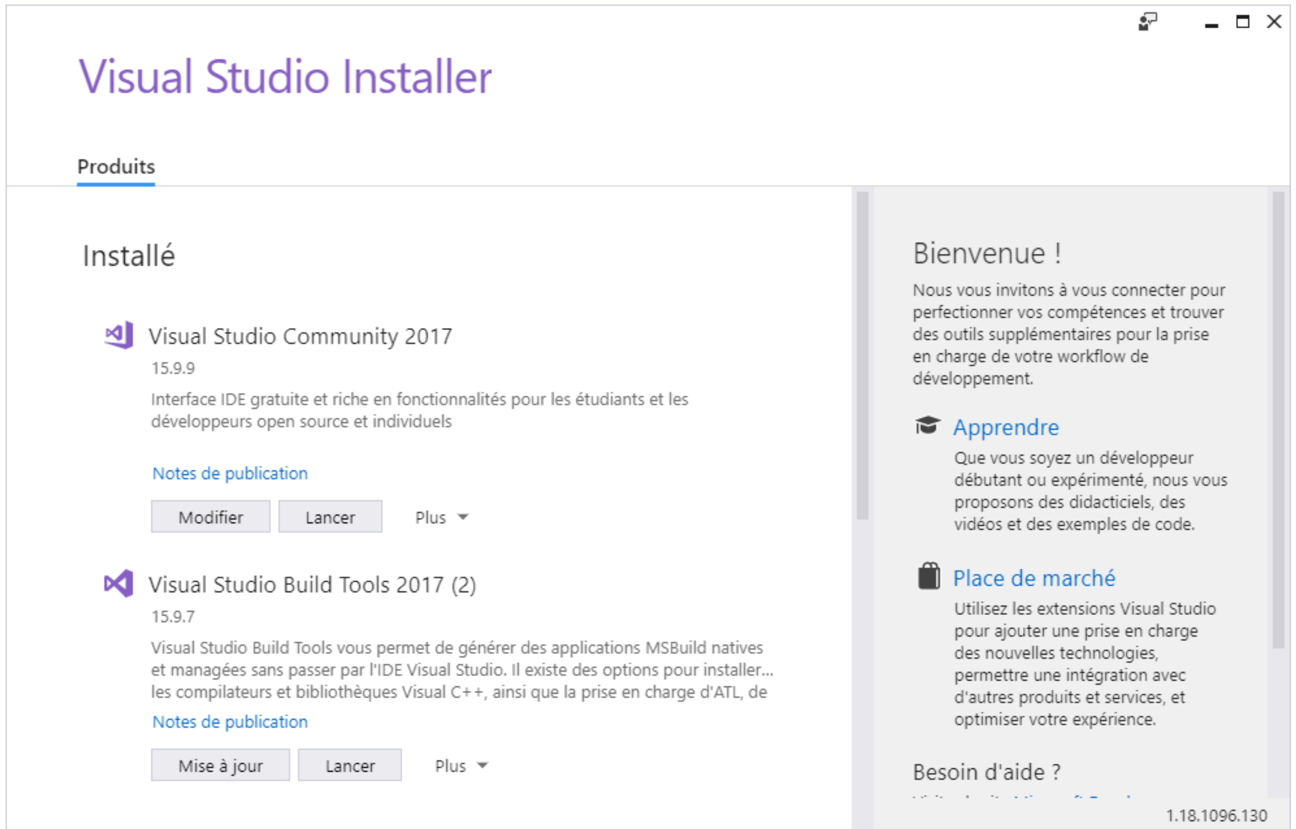


Figure 29.2. – Voici à quoi ressemble Visual Studio Installer.

Cliquez sur `Modifier -> Composants individuels`, puis descendez tout en bas chercher la ligne `GitHub Extension for Visua` Cochez-la et cela lancera son installation. Une fois fini, rouvrez Visual Studio, puis votre projet et vous verrez vous aussi la même ligne que moi.

Cliquez dessus, sélectionnez **git** et voilà, votre projet est géré par git lui aussi. Dans l'onglet **Team Explorer** en haut à droite, vous pouvez faire les mêmes opérations que nous avons vu plus haut, directement dans Visual Studio, sans passer par la console.

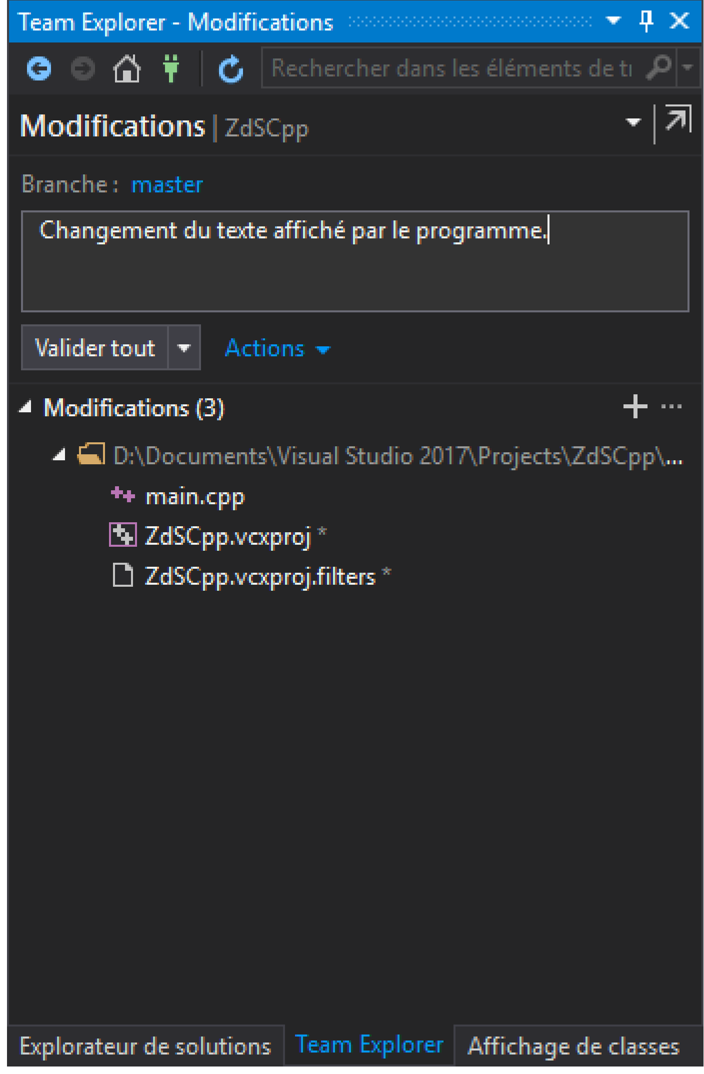


FIGURE 29.3. – Je peux ici taper mon message de commit, voir les fichiers modifiés puis valider.

Le guide officiel contient une page dédiée au sujet.

29.2. GitHub — Partager son code

29.2.1. La problématique

Notez que, dans le nom de cet outil, on retrouve Git. C'est normal, car les deux outils sont fortement liés. Si Git permet de sauvegarder l'état de son code, d'en vérifier l'historique, d'annuler ou de retravailler des *commits*, etc, il n'en reste pas moins que **votre code n'est disponible que sur votre ordinateur**. GitHub est là en complément.

GitHub est un **service en ligne** qui permet d'**héberger ses dépôts**. Puisque ceux-ci sont sur Internet, librement disponibles, vous êtes en mesure de partager votre code avec le monde entier. D'autres personnes peuvent ainsi le récupérer, le modifier et envoyer les modifications. C'est le fonctionnement et la base des projets dits **open-source**. Par exemple, Zeste de Savoir,

le site sur lequel vous lisez ce cours, utilise GitHub pour stocker son code source et permettre à plusieurs développeurs de collaborer.

Parmi les autres avantages qu'on peut citer, utiliser un service en ligne nous **protège des pannes** et autres effacements sur notre poste de travail. Si demain votre disque dur meurt, le fait d'utiliser git seul pour versionner un projet ne vous sera d'aucune aide. Par contre, s'il est hébergé sur un service en ligne, comme GitHub, il suffira de récupérer les sources et le développement peut repartir.

> **Conseil**
> Lors de la création de votre compte, utilisez la même adresse mail que vous avez utilisé lors de la configuration de git.

29.2.2. Création d'un projet

Je vais partir du principe que vous avez créé votre compte et que tout est bon. La première étape est de **créer un nouveau projet**. Pour illustrer le principe, nous allons créer un projet **Discographie**, pour permettre de sauvegarder et partager le code du T.P de la partie II du cours.

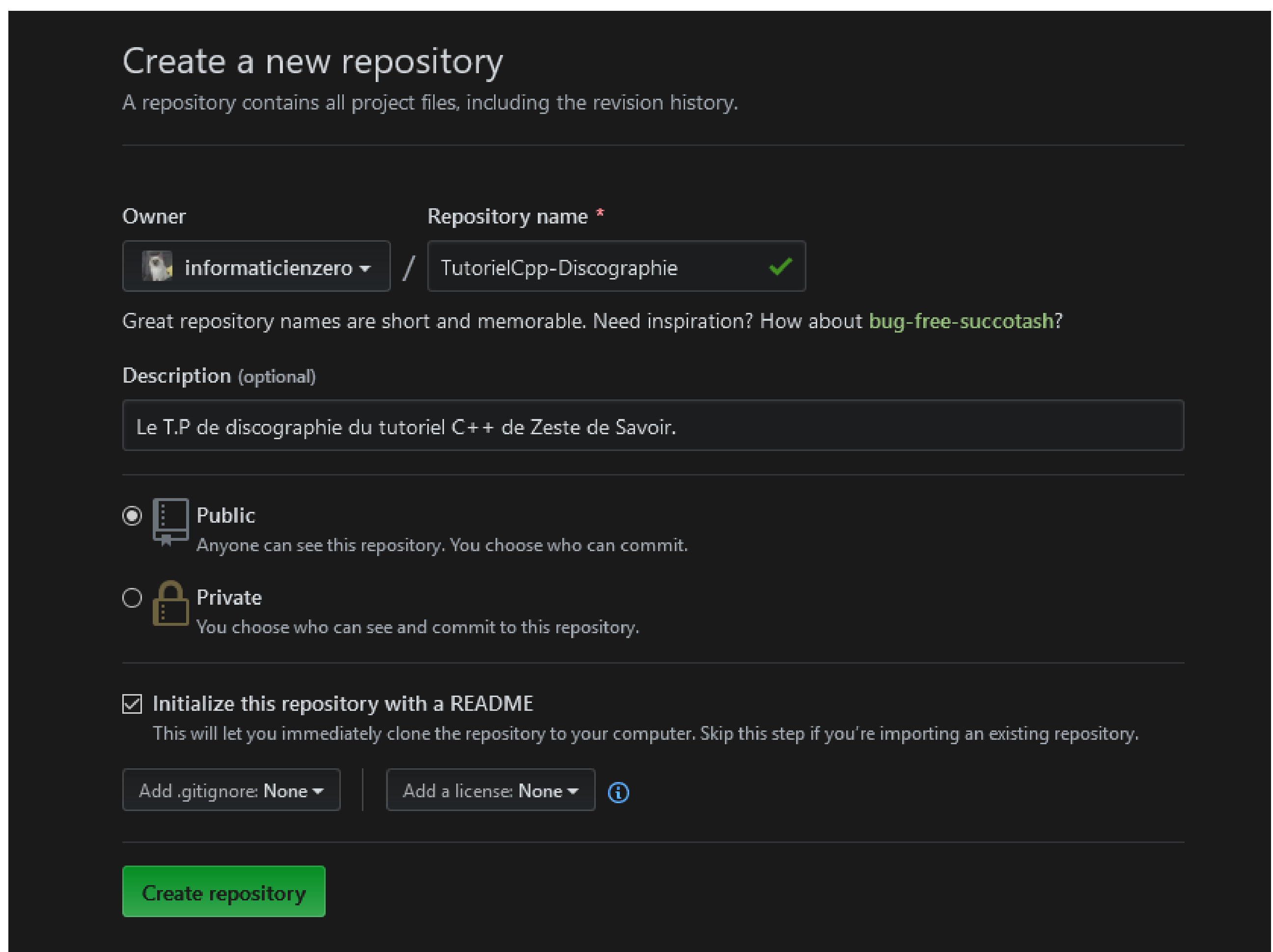


Figure 29.4. – Ma page est noire parce que j'ai un thème foncé activé, mais sinon les informations contenues dedans sont les mêmes.

Plusieurs choses sont à indiquer.

- Le **nom du projet**.
- Une **courte description**, qui indique à quoi on a affaire.
- Le choix de la **visibilité**. Public signifie que tout le monde verra votre projet, alors que *private* permet de la restreindre à vous et aux utilisateurs que vous autoriserez.
- *Initialize this repository with a README* permet de créer, en plus du projet, le fameux **fichier README**, qui contient souvent la description du projet, les aspects légaux, les informations d'installation, etc.
- On peut ajouter également un **fichier .gitignore** et **une licence**. Le premier, vous connaissez. Le deuxième indique sous quels termes vous souhaitez partager votre code. Certaines licences permettent de réutiliser le code dans des applications payantes alors que d'autres l'interdisent, etc. De très nombreuses variantes sont disponibles, mais tout ça ne nous intéresse pas dans notre cas.

Quand tout est bon, vous n'avez plus qu'à cliquer sur le bouton vert Create repository. Voici le mien .

On va partir du principe que vous avez initialisé le dépôt localement, créé un `.gitignore` et que tout est commité.

Il faut maintenant indiquer à git où se trouve le répertoire distant dans lequel nous voulons stocker le code et que nous venons de créer. Cela se fait avec la commande `git remote add origin URL`, où `URL` est l'adresse de votre dépôt. Celle-ci s'obtient en cliquant sur le bouton vert Clone or Download.

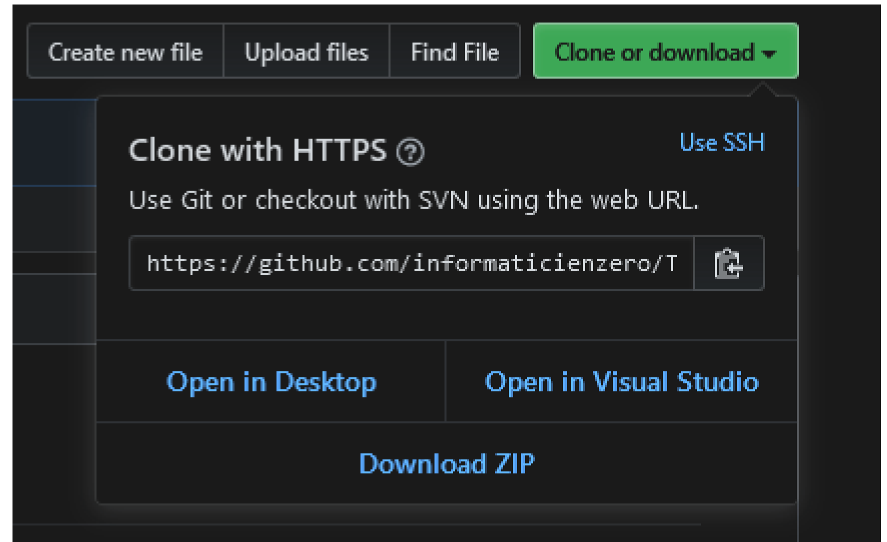


Figure 29.5. – L'URL de mon projet.

29.2.3. Récupérer localement les modifications

Maintenant qu'on est connecté avec le dépôt distant, nous sommes capables de **récupérer des modifications faites par d'autres**. Dans notre cas, personne d'autre n'a touché au projet, mais comme nous avons créé un fichier README, qui n'est pas présent localement, il faut se **synchroniser**. C'est ce que la commande `git pull origin master --allow-unrelated-histories -m "Message"` fait.

L'option `origin` désigne le dépôt distant et `master` représente la branche sur laquelle on travaille, celle par défaut. Quant au message, il résume le pourquoi de l'opération.

Dans le cas où vous travaillez à plusieurs sur le même dépôt distant, il est important de **régulièrement récupérer les modifications** des autres, pour vous assurer que vous restez à jour et pour que la gestion des conflits (deux modifications du même fichier) soit plus simple.

29.2.4. Pousser nos modifications

Il ne reste plus que l'étape consistant à **pousser nos modifications** sur le dépôt distant. Celle-ci se fait avec la commande `git push origin master`.

Actualisez votre page GitHub et vous verrez que le code est apparu. Félicitations, votre code est maintenant publiquement disponible et accessible à tous.

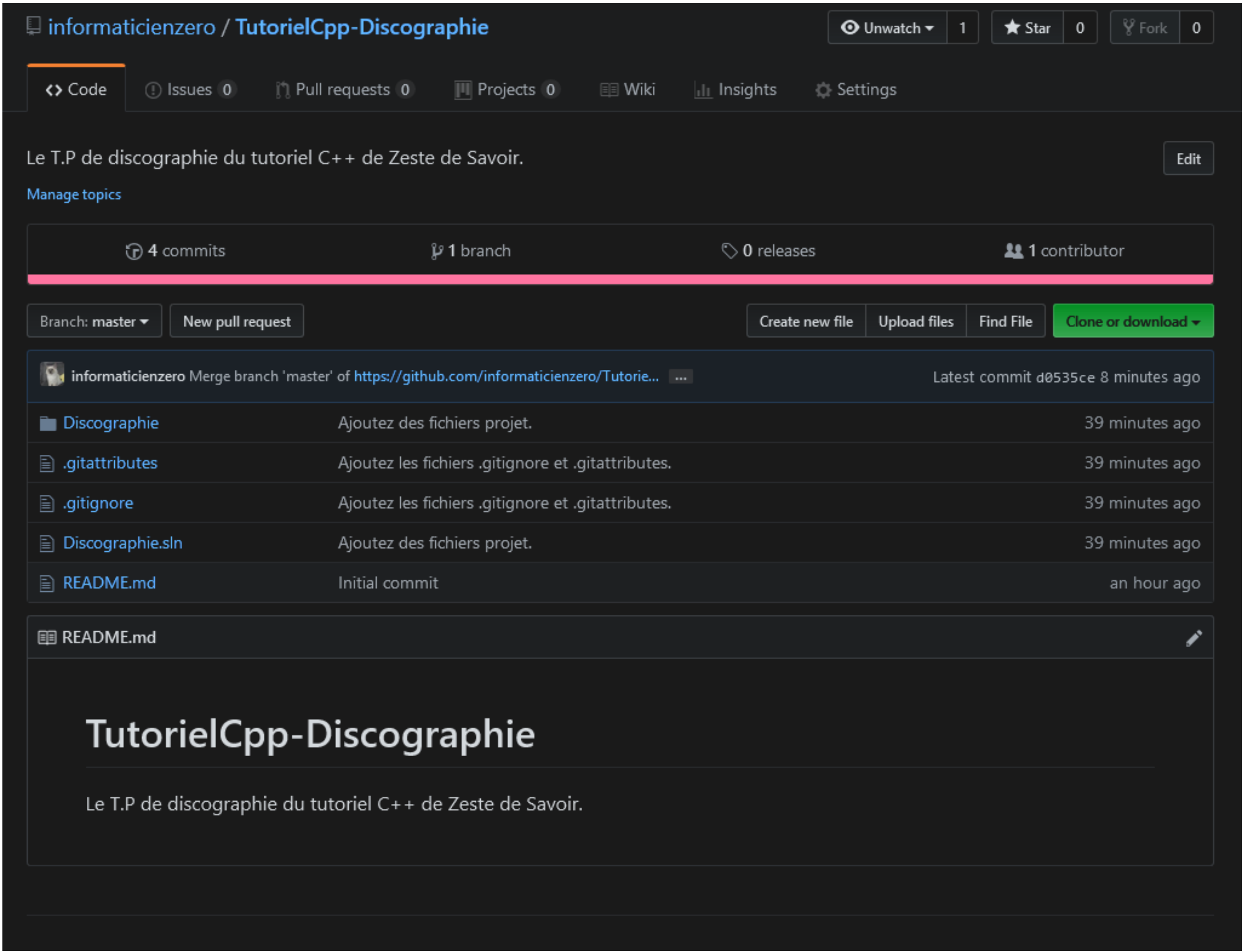


FIGURE 29.6. – Ma page GitHub après avoir poussé mes modifications.

29.2.5. Aller plus loin

Nous n'avons fait que découvrir une fonctionnalité, importante certes, mais basique de GitHub. Il vous reste encore beaucoup à apprendre. Comme ce service est très populaire, vous trouverez beaucoup de tutoriels et d'aide sur Internet. Je vous encourage à commencer par le guide officiel .

Dans celui-ci, vous apprendrez notamment à récupérer les projets des autres (on appelle ça ***forker***), à leur proposer des modifications (des ***pull requests***), à ouvrir et traiter des rapports de *bugs* (des ***issues***) et bien d'autres choses encore.

29.3. CMake — Automatiser la compilation de nos programmes

29.3.1. La problématique

Nous avons du code versionné, en ligne pour plus de sécurité et d'accessibilité et c'est déjà très bien. Mais **si je n'ai pas le même environnement** ? Dans la partie précédente, le projet posté sur GitHub a été développé avec Visual Studio. Vais-je devoir taper plusieurs lignes de

commandes pour le compiler chez moi ? C'est assez galère, avouons-le. Et dans le cas de projets complexes comme Boost, ça peut être difficile de savoir **quelles commandes écrire**.

CMake fait partie de la famille des outils permettant ce qu'on appelle en anglais du ***build automation***, soit de l'automatisation de la compilation. Il est disponible notamment pour **GNU/Linux** et **Windows**, ce qui le rend intéressant, car à partir d'un fichier d'instructions écrit dans un langage propre à CMake, l'outil est capable de générer les instructions de compilation, selon le système d'exploitation et le compilateur utilisés, sans que l'on ait à s'occuper de quoi que ce soit. Il est capable de **compiler d'autres langages que C++**, donc apprendre à l'utiliser vous sera utile même en dehors de ce tutoriel.

29.3.2. Un exemple simple

Reprenons, dans un dossier à part, nos fichiers d'en-têtes et sources qui composent le projet Discographie, puis ajoutons un fichier **CMakeLists.txt**, qui est le fichier expliquant comment compiler notre projet. La première chose à faire est de définir **quelle version minimale de CMake est autorisée**. Cela permet d'imposer l'utilisation d'une version récente, ce qui est nécessaire pour compiler en C++17.

Ensuite, autre étape importante, **le nom du projet**. Dans notre cas, il est tout trouvé.

Maintenant, il faut lister les fichiers qui constituent notre projet, en les séparant entre fichiers d'en-têtes et fichiers sources. CMake utilise des variables pour ça. On va définir, grâce à la fonction `set`, une variable `HEADERS` regroupant tous nos fichiers d'en-têtes et une variable `SOURCES` regroupant les fichiers sources.

Terminons-en indiquant comment construire une sortie nommée `Discographie`, en utilisant nos variables préalablement définies. La ligne juste au-dessus indique qu'on veut que CMake configure sa sortie pour utiliser C++17.

Au final, voilà le `CMakeLists.txt` qu'on obtient, avec commentaires en prime.

Il ne reste plus qu'à tester en lançant `cmake` avec la commande ci-dessous. Dans mon cas, sous Windows j'obtiens un fichier `.sln`, c'est-à-dire une solution Visual Studio qu'il n'y a plus qu'à ouvrir et compiler, en 32 *bits*. Pour du 64 *bits*, il faut remplacer `-A Win32` par `-A x64`

Sous GNU/Linux, on obtient un fichier Makefile, qu'il n'y a plus qu'à exécuter en lançant la commande `make`, qui va construire l'exécutable.

Dans le cas où vous avez Qt Creator d'installé, nous allons nous baser sur le compilateur fourni avec, MinGW. Pour que CMake puisse l'utiliser, il faut lancer la console inclue avec Qt. Dans mon cas, elle se trouve dans Menu Démarrer -> Qt -> 5.12.2 -> MinGW 7.3.0 (32 bit) -> Qt 5.12.2 (MinGW 7.3.0 32-bit). Le chemin est à adapter en fonction de votre version de Qt, notamment.

Maintenant, on précise à CMake que l'on veut utiliser `"MinGW Makefiles"`, ce qui va produire un fichier Makefile, comme sous l'environnement GNU/Linux.

Pour MinGW

Petite particularité concernant MinGW, il faut d'abord ajouter la ligne suivante dans le fichier `CMakeLists.txt`, avant la ligne `add_executable`.

En sortie, j'obtiens le résultat suivant. Le code est généré en lançant `mingw32-make`.

29.3.3. Lier des bibliothèques externes

Très bien, nous sommes capables de compiler notre projet peu importe la plateforme. C'est déjà une très bonne base. Mais allons plus loin en intégrant une bibliothèque externe, Boost. Je veux en effet pouvoir utiliser Boost.Log dans mon fichier `main.cpp`.

La première chose qu'il faut prendre en compte, ce sont les **différences entre Windows et GNU/Linux**. Eh oui, Boost n'est pas installé au même endroit selon le système d'exploitation utilisé. Heureusement, CMake permet de **préciser des instructions** pour un système d'exploitation ou un compilateur en particulier.

29.3.3.1. Liaison statique

Commençons par définir une variable bien utile pour la suite, qui indique qu'on veut lier statiquement Boost à notre programme. Cela se fait toujours avec le mot-clé `set`.

29.3.3.2. GNU/Linux

Ensuite, configurons les spécificités de GNU/Linux. Pour ça, rien de compliqué, un simple `if (UNIX)` suffit. Notez la présence d'un `ENDIF (UNIX)`, qui clôt le bloc conditionnel de la même manière que nos accolades en C++.

De quoi a t-on besoin ? Il existe un module qui cherche spécifiquement Boost sur le système où CMake est installé. On l'utilise comme suit, en remplaçant `composants` par les parties de Boost dont nous avons besoin.

Souvenez-vous du chapitre précédent, nous avions besoin non seulement de `log` et `log_setup`, mais également de `system` et de `thread`. La commande à écrire est donc la suivante.

29.3.3.3. Visual Studio

Rien de bien méchant pour la configuration de MSVC, le doux nom du compilateur fourni avec Visual Studio. À noter que seuls `log` et `log_setup` sont requis.

29.3.3.4. Qt Creator

Comme Qt Creator utilise notamment MinGW, on va demander à ce que CMake fasse de même. Et la première chose est de lui indiquer où se trouve l'exécutable `make`, qui permet de produire l'exécutable depuis le fichier `Makefile` généré par CMake. Ce chemin est bien entendu à adapter en fonction de l'endroit où Qt Creator est installé.

Une autre variable est à créer à cause d'un *bug* . Sans cette variable, CMake ne trouvera pas nos bibliothèques 64 *bits*. Bien entendu, cela n'est valable que si vous utilisez la version 64 *bits* de la bibliothèque.

Comme pour Visual Studio, il faut préciser le dossier contenant Boost, ainsi que les fichiers d'en-tête, puis appeler la commande `find_package`.

29.3.3.5. A-t-on trouvé Boost ?

Il est important de gérer le cas où Boost ne serait pas présent ou pas trouvé. Heureusement, c'est très simple. Quand la commande `find_package` est invoquée pour trouver une bibliothèque, elle a défini une variable `XXX_FOUND`, un booléen indiquant si oui ou non la bibliothèque demandée est présente. Dans notre cas, c'est `Boost_FOUND`.

Dans le cas où Boost est introuvable, il faut afficher un message d'erreur. La commande `message` prend justement un niveau d'erreur ainsi que le message à afficher.

Dans le cas contraire, on peut afficher les chemins des fichiers d'en-têtes et objets, par exemple. Cela se fait en affichant `Boost_INCLUDE_DIRS` et `Boost_LIBRARY_DIRS`.

29.3.3.6. Inclure les fichiers nécessaires

Ces deux variables ne sont pas juste là pour être affichées. Pour que notre programme soit correctement généré, il faut qu'on lui indique les fichiers d'en-têtes et objets à inclure. C'est justement ce que font les commandes `include_directories` et `link_directories`.

29.3.3.7. La construction de l'exécutable

Comme dans l'étape précédente, nous voulons configurer le programme pour qu'il soit compilé avec C++17. Les deux commandes vues plus haut, dans l'exemple simple, ne changent donc pas.

Il ne reste plus qu'à lier Boost à notre exécutable. Cette étape nécessite elle aussi une séparation entre Windows et GNU/Linux. En effet, la façon d'ajouter les bibliothèques n'est pas la même et GNU/Linux réclame une option supplémentaire.

Dans les deux cas, on utilise la commande `target_link_library`. Pour Windows, c'est simple, il suffit de donner le chemin des fichiers objets, c'est-à-dire `Boost_LIBRARY_DIRS`. Pour GNU/Linux, il faut rechercher une bibliothèque supplémentaire, `-lpthread`, puis préciser quelles bibliothèques sont précisément requises.

> *i* Parallèle
>
> Ce qui se passe ici est en fait la même chose que ce que nous faisons manuellement au chapitre précédent.

Je vous mets donc les commandes nécessaires ci-dessous.

29.3.3.8. Fichier final

Voici le fichier au final, commenté et complet.

29.3.3.9. Exécuter CMake

Maintenant que le fichier `CMakeLists.txt` est prêt, alors il ne nous reste plus qu'à lancer la génération de notre programme. Voici comment je construis le programme avec Boost 32 *bits*. J'obtiens une solution **.sln** prête à être compilée et exécutée.

Sous GNU/Linux, j'obtiens un fichier `Makefile` qu'il ne reste plus qu'à lancer en appelant `make`. J'obtiens au final mon programme fonctionnel.

Avec MinGW, pareil, je relance une console Qt et j'exécute la commande `cmake` avec les bons paramètres.

29.3.4. Définir des variables au lancement

Typiquement, pour la partie MinGW, j'ai défini en dur la variable `Boost_ARCHITECTURE` comme valant `"-x64"`. Mais si jamais on veut compiler en 32 *bits*? Le mieux serait de ne pas avoir cette variable en dur, mais plutôt de **la définir quand on en a besoin**. C'est justement ce que nous permet de faire CMake.

L'option `-DNOM` permet de définir une variable `NOM` avec la valeur de notre choix. On peut ainsi préciser explicitement `"-x64"` quand on le veut, sans être forcé comme avant.

29.3.5. Aller plus loin

CMake est un outil puissant, qui permet de gagner en souplesse, mais le maîtriser demande bien plus que de savoir les quelques commandes que nous avons vues ici. De nombreuses ressources sont disponibles, à commencer par le cours officiel sur le site de CMake. Vous pouvez également jeter un œil à ce tutoriel en français .

Par exemple, il est possible de **lancer automatiquement Doxygen** en même temps que la compilation, afin de disposer également de la documentation du projet. Vous pouvez aussi apprendre des techniques pour ne pas avoir à **mettre à jour à la main la liste des fichiers** d'en-têtes et sources. Mais tout ça, c'est de votre ressort maintenant.

29.4. Aller plus loin

Si je devais vous présenter tous les outils utiles à un développeur C++, ce tutoriel serait deux fois plus gros minimum et ce chapitre bien indigeste. Nous en avons déjà vu trois, importants et très largement répandus. Mais il en existe encore **de nombreux autres**, que vous serez amenés à utiliser (ou pas) au fur et à mesure que vous progresserez.

29.4.1. TravisCI — De l'intégration continue

Imaginez qu'à chaque *commit* sur GitHub, votre code soit automatiquement compilé, les tests unitaires lancés et la documentation automatiquement générée et mise à jour ? C'est ce que permet de faire TravisCI et c'est ce qui s'appelle **l'intégration continue** . En plus, l'outil est **gratuit** si votre projet est open-source.

29.4.2. CppCheck — Vérification du code

Un autre outil qui rend de fiers services, j'ai nommé CppCheck . Cet outil effectue de **nombreuses vérifications** sur votre code pour vous aider à le rendre encore plus robuste. On peut citer notamment la vérification des limites de tableaux, les divisions par zéro, ou encore la vérification des exceptions lancées.

Beaucoup d'informations, mais aucun problème trouvé sur le projet Discographie. Ouf !

29.4.3. StackOverflow — La réponses à quasiment toutes les questions

S'il y a un site que tous les développeurs connaissent et lisent, c'est bien StackOverflow . C'est **LE site de questions-réponses** sur Internet. Si vous n'avez aucune réponse à votre problème sur StackOverflow, alors vous n'avez vraiment pas de chance du tout. Peu importe le langage, peu importe la technologie, tout est abordé sur ce site. Mettez-le dès maintenant en favori, il va vous sauver la vie !

29.4.4. En résumé

- Le gestionnaire de code source git permet de sauvegarder et versionner son code source. Il permet ainsi de tracer un historique du projet, de revenir à des versions antérieures et de développer plusieurs fonctionnalités en parallèle.
- Le service en ligne GitHub permet de partager son code source aux autres, permettant ainsi une sauvegarde pérenne, la création de *bugs*, la proposition de correctifs ou d'améliorations, entre autres choses.
- CMake automatise la génération du code, permettant, avec un unique fichier de configuration, de générer des projets Visual Studio, des *Makefiles* et bien d'autres choses encore.
- Chacun de ces outils offre de nombreuses possibilités que ce cours ne peut pas aborder et qu'il convient au lecteur de découvrir par lui-même.
- De nombreux outils pour C++ sont disponibles, qu'un développeur découvre et utilise au fil du temps.

Finalement, être un développeur, ça recouvre bien plus que simplement écrire du code. Tous ces outils sont là pour nous faire gagner du temps et de l'efficacité. Bien entendu, tous les développeurs C++ n'utilisent pas tous les mêmes outils. Certains vous seront peut-être imposés, en fonction du projet sur lequel vous travaillerez. Mais certains principes restent universels.

— La documentation est votre meilleure amie.
— Une bonne documentation épargnera beaucoup de peine à ceux qui passeront derrière vous ou qui utiliseront vos produits.
— On crée tout le temps des *bugs*, alors apprendre à les détecter et les éliminer est une bonne habitude. On essayera bien entendu d'en éliminer un maximum dès la compilation, mais pour ceux qui restent, le *débogueur* vous attend.
— Comprendre la compilation est utile, parce qu'en comprenant mieux ce qui se passe sous le capot, vous pouvez optimiser (comme avec `constexpr`) et mieux comprendre les erreurs (erreur de *linkage* par exemple).

Le fait est qu'en terminant cette partie, vous avez bien progressé. Vous commencez à avoir la mentalité et les pratiques d'un bon développeur. Nous allons justement mettre l'ensemble du cours en pratique avec une étude de cas.

Contenu masqué

Contenu masqué n°72 :
Fichier .gitignore pour Visual Studio

```

```

Retourner au texte.

Contenu masqué n°73 :
Fichier .gitignore pour GNU/Linux

```

```

Retourner au texte.

Cinquième partie

[EdC] Zesty Creatures, un Pokémon-like en console !

Sixième partie

La Programmation Orientée Objet

Ce cours s'achève ici, mais votre apprentissage du C++ est loin d'être terminé. Il reste encore beaucoup de choses à apprendre. Vous pouvez commencer par les contenus C++ disponibles sur Zeste de Savoir. Certains approfondissent des concepts que nous avons introduits dans ce tutoriel, d'autres vous présenteront des notions nouvelles. Enfin, pour continuer votre apprentissage, voici quelques derniers conseils.

- **Soyez curieux**: fouillez sur Internet pour découvrir de nouvelles méthodes, approfondissez celles que vous connaissez, renseignez-vous, testez de nouveaux outils, etc.
- **Entrainez-vous**: c'est le meilleur moyen de progresser. Faites des projets qui vous tiennent à cœur, mettez en œuvre des algorithmes connus, réalisez des exercices, etc.
- **Lisez des codes produits par d'autres personnes**: découvrez comment elles procèdent, apprenez d'elles de nouvelles techniques ou façons de faire et progressez en suivant leurs conseils. Vous pouvez par exemple commencer en visitant les forums de ce site.
- Enfin, le plus important: **amusez-vous**!

Liste des abréviations

ASCII American Standard Code for Information Interchange. 153

ISO International Organization for Standardization. 19, 23

PGCD Plus Grand Commum Diviseur. 2, 83, 92, 249

UB Undefined behavior. 101, 292