UNIVERSITÉ DE GENÈVE

Cours C++

Jacques.Menu@unige.ch - 1er février 2004
http://cui.unige.ch/DI/cours/SystInformatiques

1 Introduction

☞ Ce cours est incomplètement mis à jour depuis une version ancienne basée sur C++ 2.2, et il est encore encore en cours d'évolution (automne 2003) !

1.1 Le langage C++

C++ est une **extension presque stricte de C** définie par Bjarne Stroustrup pour ATT. On peut qualifier ce langage en quelques mots:

- *la partie 'C' du langage est très voisine de ANSI C*, dont la conception a d'ailleurs été influencée par C++;
- on peut *passer graduellement de C à C++*: il suffit en premier lieu de changer de compilateur, sans nécessairement utiliser les nombreuses possibilités complémentaires qu'offre C++. C''est ce que `[Cox 88]` appelle une *approche évolutioniste plutôt que révolutionnaire* (*evolutionary, not revolutionary*);
- les possibilités nouvelles par rapport à C *ne 'coûtent rien'* dans un programme qui ne les utilise pas;
- le premier intérêt du passage à C++ est de bénéficier du **typage fort**, permettant beaucoup plus de contrôles statiques des programmes à la compilation, et de la gestion de **références** au sens de Pascal;
- le but final toutefois est de profiter pleinement de la puissance de la **programmation orientée objets** (object oriented programming en anglais) pour le développement de logiciels.

La partie 'objets' de C++ est fortement inspiré de **Simula 67**, qui fut le précurseur de l'orientation objets. *Pratiquement tous les concepts s'y trouvaient déjà, sauf l'héritage multiple*. Le fait est assez remarquable: il n'est pas fréquent, surtout en informatique, d'avoir 20 ans d'avance sur son temps !

Tous les exemples illustrant ce support ont été testés avec GCC 3.3 et la librairie standard GNU C++ en version 3 sur différentes architectures matérielles et logicielles.

☞ Pour les conventions de nomination des fichiers, nous nous en tenons à `'.h'` et de `'.cc'`.

Bien que C++ soit une extension de C, **nous ne présentons pas le langages C en tant que tel**: toute personne désirant apprendre C++ nous semble en effet devoir **étudier directement C++** pour tous les avantages qu'il apporte par rapport à C.

Citons encore que la seule référence faisant foi pour nous est la définition du langage C++ par B. Stroustrup pour ATT 2.1 `[Ellis & Stroustrup 90]`. Certains livres sur le langages font référence à la version 1, dont certaines parties sont maintenant obsolètes.

1.2 Orientation de ce cours

C++ est **un langage très riche**, ce qui rend certains aspects parfois difficiles à maîtriser d'emblée. Nous avons donc choisi de diviser ce support en trois parties principales, soit:

- la présentation du *langage de base*, qui offre des possibilités analogues à celles de C, mais avec des avantages spécifiques;
- la présentation des possibilités de *programmation orientée objets*, qui sont sans doute les plus riches disponibles dans un seul et même langage de grande diffusion à l'heure actuelle;
- une dernière partie regroupant des *points avancés du langage*, ce qui est aussi l'occasion de revenir sur certains aspects traités dans les deux premières parties et de les discuter plus en détail.

Tous les *exemples* sont fournis *complets avec le résultat de leur exécution*. De plus, un effort particulier a été fait pour que tous les identificateurs laissés au choix du programmeur soient *en français*, afin de ne pas gêner ceux qui ne maîtrisent pas l'anglais. Enfin, les expressions les plus fréquentes sont données dans ces deux langues.

Ce cours *pré-suppose une connaissance de la programmation* dans un ou plusieurs langages.

1.3 Bibliographie

Elle est très importante autour de C++, mentionnons simplement :

[Ellis & Stroustrup 90] Ellis M.A & Stroustrup B., "The annotated C++ reference manual"
Addison-Wesley, 0-201-51459-1, 1990

La référence sur C++, avec des considérations sur les choix de conception du langage.

[Ray Lischner 2003] Ray Lischner, "C++ in a Nutshell"
O'Reilly, 0-596-00298-X, 2003

Présentation détaillée du langage C++,, illustrée par des exemples.

[Birtwistle & al. 73] Birtwistle G. & al., "Simula Begin"
Van Nostrand Reinhold, 0-905897-37-4, 1973

Le langage Simula 67, précurseur de l'approche objets.

[Dewhurst & Stark 89] Dewhurst S. & Stark K., “Programming in C++“
Prencice Hall Software Series, 0-13-723156-3, 1989

Bonne introduction à la puissance d’expression de C++.

[Dugerdil 90] Dugerdil Ph., “Smalltalk-80“
Presses Polytechniques Romandes, 2-88074-182-3, 1990

Bonne présentation du langage Smalltalk, où tout est objet, illustrée par des exemples.

[Menu 92] Menu J., “Langages informatiques : concepts et exemples“
Support de cours polycopié
Laboratoire de Compilation
Ecole Polytechnique Fédérale de Lausanne, 1992

Présente les langages informatiques du point de vue de la sémantique comparée. Contient une présentation de l’évolution de la programmation orientée objets de Simula 67 à Smalltalk 80 en passant par C++, des exemples de programmation par contraintes, ainsi que le détail des algorithmes classiques de gestion de la mémoire.

[Meyer 88] Meyer B., “Object-oriented software construction“
Prentice Hall, 0-13-629049-3, 1988

Ouvrage de référence. Intéressante discussion sur les outils et les projets logiciels.

UNIVERSITÉ DE GENÈVE

2 Langage de base

Ce chapitre présente le sous-ensemble de C++ équivalant en gros à ce qu'offre C. L'intérêt de C++ à ce niveau réside dans les possibilités d'expression plus fine et les contrôles intensifs effectués par les compilateurs.

2.1 Salut tout le monde!

Une tradition veut que le tout premier programme que l'on écrit en C soit `HelloWorld` ('salut tout le monde', en français). Pour ne pas manquer aux bons usages, voici donc ce programme dans une version C++ :

```
// Le programme 'Salut tout le monde' en C++

#include <iostream>     // on utilise la gestion des flots


/*
Ce commentaire multi-ligne précède
la fonction 'main', qui est toujours
le programme principal
*/

int main ()
     {
     std::cout << "Salut tout le\n" "monde" << "\n";// et voila!
     }
```

Le résultat produit est :

```
Salut tout le
monde
```

Précisons quelques aspects du langage visibles sur cet exemple :

- les **commentaires** ont *deux formes possibles*, soit à droite de // et jusqu'à la *fin de la ligne*, soit entre `/*` et `*/` indépendamment des fins de lignes.
  On conseille de réserver ces derniers à la '*mise hors course*' de pans entiers de programmes, car ils ne peuvent pas être imbriqués ;
- pour **utiliser des déclarations placées dans un fichier** '`.h`' ou sans le '`.h`' comme c'est le cas de la librairie standard C++, on l'inclut par :

  ```
  #include <…>
  ```

  s'il s'agit d'un *fichier fourni par l'implantation utilisée*, ou par :

  ```
  #include "…"
  ```

s'il s'agit d'un fichier qui a été écrit en complément à ceux fournis.

La différence réside dans les dossiers qui sont fouillés pour y trouver le fichier correspondant ;

- il faut une fonction **`main`** du type **`int`**, avec ou sans paramètres, qui joue le rôle de *programme principal* : c'est elle qui est exécutée lorqu'on lance le programme, à charge pour elle d'appeler d'autres fonctions à volonté;
- le corps d'une fonction est parenthésé par **`{`** et **`}`** ;
- **chaque instruction se termine par** '**`;`**', qui joue le rôle de *terminateur* d'instruction, et non de séparateur comme en Pascal par exemple ;
- les **chaînes de caractères** sont parenthésées par des guillements '**`"`**'. Certains **caractères non-imprimables** comme '*fin de ligne*' et d'autres utilisés spécifiquement par C++ s'écrivent respectivement :

```
\t      tabulateur horizontal
\n      fin de ligne
\f      fin de page
\\      le caractère '\'
\?      le caractère '?'
\'      le caractère "'"
\"      le caractère '"'
```

Ainsi:

```
std::cout << "J\"aime";
```

produit comme résultat:

```
J"aime
```

- les chaînes de caractères adjacentes sont *concaténées* ;
- les entrées/sorties textuelles utilisent des **flots** ('*stream*' en anglais). **`std::cout`** est le flot standard de sortie, et on utilise l'opérateur **`<<`** pour *cascader des écritures* successives sur ce flot, comme dans :

```
cout << "Salut tout le monde" << "\n";
```

`std::cout` est connecté au *clavier* lors d'un emploi interactif du programme.

On peut utiliser le caractère *souligné* '_' pour aérer les identificateurs: ce caractère compte alors comme une lettre.

☞ En C++, **les lettres majuscules et minuscules sont considérées comme distinctes**, et les identificateurs `bonjour`, `Bonjour` et `BONJOUR` sont trois identificateurs *différents*.

☞ **Tous les caractères** constituant les identificateurs C++ sont **significatifs**.

☞ **`std`** est un espace de noms (*name space*), auquel appartient `cout`.

La mise en pages est libre: on peut utiliser des tabulateurs, espaces, lignes blanches et commentaires à volonté pour aider à la lisibilité des textes sources.

2.2 Types de base

C++ fournit des **types de base non-structurés** (*'fundamental types'* en anglais) permettant de représenter des nombres entiers et des nombres flottants.

On dispose de **types entiers signés** (`signed` en anglais), implantés en **complément à 2** ("*2's complement*" en anglais), et qui peuvent donc déborder (*'to overflow'* en anglais):

- `short int`, `int` et `long int` sont des types entiers dont la taille en nombre de bits dépend de l'implantation ;
- le type `int` est celui des entiers *'naturels'* sur l'architecture où l'on se trouve, 32 bits le plus souvent ;
- les `long int`, que l'on peut abbréger en `long`, sont d'une taille *supérieure ou égale* à celle des `int` ;
- les `short int`, que l'on peut abbréger en `short`, sont d'une taille *inférieure ou égale* à celle des `int`.

Le mot clé `signed` pourrait être utilisé comme dans `signed long int`, mais il est en fait implicite pour les types entiers.

On peut obtenir la taille des informations manipulées par C++ à l'aide de l'opérateur `sizeof`, qui retourne **la taille en octets d'un type ou d'une expression** passée en paramètre, auquel cas cette expression n'est en fait *pas évaluée*. Le type retourné par `sizeof` est `size_t`, défini dans le fichier `StdDef.h`, fourni par l'implantation.

On peut résumer la sémantique des préfixes `short` et `long` par:

```
sizeof (short int) <= sizeof (int) <= sizeof (long int)
```

On dispose aussi en C++ de **types entiers non signés** (`unsigned` en anglais), implantés en complément à 2^n, où `n` est le nombre de bits utilisés, ce qui fait qu'ils ne peuvent pas déborder. Les types correspondants sont:

- `unsigned short int`
- `unsigned int`
- `unsigned long int`

et sont tels que:

```
sizeof (unsigned short int) <=
    sizeof (unsigned int) <=
        sizeof (unsigned long int)
```

C++ fournit aussi **deux types caractères,** permettant de stocker le **code numérique d'un caractère** du jeu utilisé sur l'implantation condidérée:

- `signed char` est le type des caractères représentés par un code entier signé;
- `unsigned char` est le type des caractères représentés par un code entier non signé;

☞ On a toujours:

```
sizeof (char) = sizeof (unsigned char) = 1
```

Chaque implantation décide lequel des deux types `signed char` et `unsigned char` est en fait choisi lorsqu'on utilise simplement le type **`char`** sans préfixe. Dans le cas de MPW C++, il s'agit de `signed char`, qui contient les codes ASCII des caractères.

☞ Le type **caractère** n'est qu'un type entier particulier en C++: il se prête donc à toutes les opérations arithmétiques usuelles.

Le fichier **`Limits.h`** contient des constantes définissant les **limites des représentations des nombres entiers** sur une implantation donnée. Dans le cas de MPW C++, il contient:

```
/*
     Limits.h -- Sizes of integral types

     Copyright Apple Computer,Inc.1987, 1990
     All rights reserved.
*/

#ifndef __LIMITS__
#define __LIMITS__

#define CHAR_BIT        8
#define CHAR_MAX        127
#define CHAR_MIN        (-128)
#define MB_LEN_MAX      1
#define INT_MAX         2147483647
#define INT_MIN         (-2147483648)
#define LONG_MAX        2147483647
#define LONG_MIN        (-2147483648)
#define SCHAR_MAX       127
#define SCHAR_MIN       (-128)
#define SHRT_MAX        32767
#define SHRT_MIN        (-32768)
#define UCHAR_MAX       255U
#define UINT_MAX        4294967295U
#define ULONG_MAX       4294967295U
#define USHRT_MAX       65535

#endif
```

C++ fournit encore trois **types de nombres en virgule flottante** (real en Pascal):

- **`float`**
- **`double`**
- **`long double`**

tels que:

```
précision (float) <= précision (double) <= précision (long double)
```

Le type **double** joue un rôle particulier, car c'est par défaut celui des constantes flottantes, et *tous les arguments flottants sont convertis en double lors du passage des paramètres*.

Le dernier type de base défini par C++ est **void**, qui est le type de toute **valeur inexistante**. Ainsi:

```
void une_procedure ()
     {
     // des instructions
     }
```

définit une **procédure** au sens de Pascal. **void** est utile pour indiquer le type retourné par une fonction, comme dans l'exemple ci-dessus, ainsi que pour définir un type pointeur universel **void ***.

On peut utiliser **typedef** pour créer un **synonyme d'un type** décrit au passage, comme dans:

```
typedef int mon_type_entier;
```

qui définit `mon_type_entier` comme un synonyme du type `int`. D'autres exemples plus significatifs seront rencontrés plus loin.

On n'a pas en C++ de type **booléen** prédéfini. On trouve en fait dans le fichier **Types.h** sur Macintosh la spécification:

```
enum {false, true};
typedef unsigned char Boolean;
```

qui définit les deux constantes `false` et `true`, ainsi que le type `Boolean` comme synonyme de `unsigned char`.

Il est bien sûr de créer un tel fichier dans une implantation qui ne fournit pas de définition du type booléen. Nous reverrons le mot clé `enum` en détail au paragraphe 2.13, page 23.

Le type **Boolean** n'est lui aussi qu'un type entier particulier en C++: il se prête donc à toutes les opérations arithmétiques usuelles, ce qui peut surprendre les habitués de langages comme Pascal, Modula-2 ou Ada.

2.3 Constantes

Comme constantes des types de base non structurés, le langage C++ founit des constantes entières, de caractères et de chaîne de caractères.

Les **constantes entières** sont de la forme de base:

```
12        la valeur 12 en décimal
012       la valeur 12 en octal, soit 10 en décimal
0x12      la valeur 12 en hexadécimal, soit 18 en décimal
0X12      la valeur 12 en hexadécimal, soit 18 en décimal
0xFf      la valeur Ff en hexadécimal, soit 255 en décimal
```

Dans le cas des constantes octales et hexadécimales, le premier caractère est le chiffre zéro.

De plus, les constantes entières peuvent avoir un suffixe indiquant le type de la constante:

- si une constante `décimale` n'a pas de suffixe, elle est du premier dans l'ordre des types:

  ```
  int, unsigned int, long int, unsigned long int
  ```

  capable de contenir sa valeur.
- si une constante *octale* ou *hexadécimale* n'a pas de suffixe, elle est du premier dans l'ordre des types:

  ```
  int, long int, unsigned long int
  ```

  capable de contenir sa valeur.
- lorqu'un constante entière est suffixée par **`u`** ou **`U`**, elle est du premier dans l'ordre des types:

  ```
  unsigned int, unsigned long int
  ```

  capable de contenir sa valeur.
- lorqu'un constante entière est suffixée par **`l`** (lettre "ell'") ou **`L`**, elle est du premier dans l'ordre des types:

  ```
  long int, unsigned long int
  ```

  capable de contenir sa valeur.
- enfin, toute constante entière suffixée par les deux lettres **`U`** et **`L`**, en majuscules ou en minuscule, dans cet ordre ou en ordre inverse, est du type:

  ```
  unsigned long
  ```

Les **constantes de caractères** sont de la forme:

```
'x'       le caractère "x"
'\n'      fin de ligne (13 décimal en ASCII)
'\t'      tabulateur (9 décimal en ASCII)
'\xD'     'D' en hexadécimal, 13 en décimal
'\0'      caractère dont le code est nul
```

La valeur d'une constante de caractère est le code du caractère dans l'implantation considérée. Si elle dépasse la valeur la plus grande du type `char`, elle dépend alors de l'implantation.

C++ supporte aussi des *constantes multicaractère*, du type `int`, pour des codes plus étendus que le code ASCII, mais cela est dépendant de l'implantation. Une telle constante s'écrit par exemple `'xB'`.

Une constante de caractère précédée du préfixe **`L`**, comme `L'1a'`, est un *caractère large* ('*wide character*' en anglais), et elle est du type **`wchar_t`**, défini dans le fichier **`StdDef.h`**. Cette possibilité existe pour les codes de caractères non représentables sur un seul octet, comme Unicode qui en cours de définition.

Les **constantes de chaîne** ont été décrites au paragraphe 2.1, page 1.

Les chaînes C++ sont **implicitement terminées par un caractère de code nul** (`\0`). Ce caractère sert ensuite à reconnaître la fin d'une chaîne, et il est ajouté à la compilation *après* concaténation de chaînes adjacentes éventuelles.

Certaines fonctions de traitement de chaînes de la librairie `stdlib.h`, comme `strcpy`, ajoutent également ce caractère à la fin des chaînes de caractères qu'elles produisent.

Enfin, les **constantes flottantes** sont de la forme:

```
123.4
1234e-1
1234E-1
.1234e3
```

Les remarques suivantes s'appliquent aux constantes flottantes:

- la partie entière ou la partie décimale peut être absente, mais pas les deux à la fois;
- le point décimal ou la partie exposant peut être absente, mais pas les deux à la fois;
- l'exposant est un entier pouvant être signé.

Le **type d'une constante flottante** est `double` par défaut, sauf si on utilise un suffixe:

- `f` et `F` indiquent le type `float`;
- `l` et `L` indiquent le type `long double`.

2.4 Variables non-modifiables

C++ offre la gestion de **variables non-modifiables après leur initialisation** ('*read-only*' en anglais). On utilise pour celà le mot clé réservé `const`, comme dans:

```
const int taille = 5;
```

L'initialisation de telles variables est obligatoire lors de leur définition.

Nous verrons l'emploi du mot clé `const` dans les passages de paramètres au paragraphe 2.25, page 48. Le Aspects avancés du langage contient des exemples plus complexes, notamment dans le cas des pointeurs passés en paramètres.

2.5 Fonctions

Une fonction C++ peut avoir ou ne pas avoir d'arguments selon les besoins. **Les arguments peuvent être de n'importe quel type, de même que la valeur retournée par une fonction**. Rappelons que le type `void` indique une *procédure*, soit une fonction ne retournant pas de valeur.

L'instruction:

```
return;
```

provoque la sortie du corps de la procédure où elle apparaît, et le retour à l'appeleur de cette procédure. Il y a un `return;` *implicite* à la fin du corps d'une procédure.

Dans le cas d'une fonction retournant une valeur, on utilise:

```
return une_expression;
```

pour retourner la valeur `une_expression` à l'appeleur de la fonction. On doit toujours avoir une sortie explicite d'une fonction par `return une_expression;`, sans quoi on ne saurait quelle valeur doit être retournée.

C++ permet de spécifier qu'une fonction doit être traitée autant que possible comme une *macro-instruction*, en créant une copie de son code en remplacement de chaque appel, tout en faisant malgré tout les contrôles sémantiques de ces appels. Cela se fait au moyen du mot clé **`inline`**, comme dans:

```
inline int carre (int n)
     { return n * n; }
```

Il n'y a toutefois *aucune garantie* que cette directive sera appliquée par le compilateur. Cela peut en particulier ne pas être le cas si la fonction '`inline`' contient un passage de paramètre par référence ou une instruction de boucle comme `for`, `while` ou `do`.

La définition et l'appel d'une **fonction sans paramètres** se fait en indiquant une liste d'arguments vide entre parenthèses, comme dans:

```
int f ()
     {
     // … … … …
     }

//   … … … …

cout << f ();
```

Les parenthèses sont *nécessaires à l'appel* car le nom de la fonction est lui-même un opérande founissant un pointeur sur son code. Dans le cas où l'on oublie ces parenthèses, le compilateur fournit un avertissement.

C++ propose deux modes de passages de paramètres:

- le **passage par référence**, caractérisé par le marqueur syntaxique **`&`**, analogue à celui de Pascal et Modula-2:

  ```
  carre (int & n)
  ```

- le **passage par valeur**, dans tous les autres cas.

Le détail des passages de paramètres est présenté au paragraphe 2.25, page 48.

Voici un exemple contenant la définition d'une fonction et des appels à elle:

```
#include <stream.h>

int SommeCarres (int borneInf, int borneSup)
     {
```

```
        if (borneInf > borneSup)
             return 0;

        else
             return
                  borneInf * borneInf
                       +
                  SommeCarres (borneInf + 1, borneSup);
        }


main ()
        {
        cout << ("%d", SommeCarres (1, 5));
        }
```

Ce programme produit le résultat:

```
55
```

2.6 Références

C++ offre une gestion de références allant *au-delà du simple passage de paramètres par référence*. Voici ce qu'il en est:

- une référence est **un alias, un synonyme, pour la variable référencée**;
- toutes les occurences de la référence sont **implicitement déréférencées** pour accéder à la variable référencée;
- une fois initialisée pour référencer une variable donnée, **une référence ne peut plus être changée pour référencer une autre variable**, puisqu'il s'agit d'un synonyme pour la première;
- une référence, lorsqu'elle est déclarée comme **variable locale**, doit obligatoirement être **initialisée sur sa déclaration**.

La notion de référence est aussi utilisée pour le **passage de paramètres par référence** de manière analogue à ce qui se fait en Pascal.

Si on initialise une référence avec une valeur et non une variable, une **variable anonyme** est *créée et initialisée avec cette valeur*, et la référence devient un alias pour cette variable anonyme.

Voici un exemple illustrant l'emploi de références en C++:

```
#include <stream.h>

long carre (long &n)
        { return n * n; }
```

```
main ()
     {
     long          un_entier_long = 9;
     typedef       long & long_ref;

     long_ref      reference_1 = 45;
                   // creation d'une VARIABLE ANONYME
     cout << "reference_1 = " << reference_1 << "\n";

     reference_1 = un_entier_long;
                   // cette VARIABLE ANONYME change de valeur
     cout << "reference_1 = " << reference_1 << "\n\n";

     long_ref      reference_2 = un_entier_long;
                   // "reference_2" est un alias pour "un_entier_long"
     cout << "reference_2 = " << reference_2 << "\n";

     reference_2 += 17;
                   // "un_entier_long" change de valeur
     cout << "reference_2 = " << reference_2 << "\n";
     cout << "un_entier_long = " << un_entier_long << "\n\n";

     un_entier_long = carre (un_entier_long);
                   // "un_entier_long" change de valeur
     cout << "un_entier_long = " << un_entier_long << "\n";

     cout << "reference_2 = " << reference_2 << "\n";
     }
```

Les résultats sont:

```
reference_1 = 45
reference_1 = 9


reference_2 = 9
reference_2 = 26
un_entier_long = 26


un_entier_long = 676
reference_2 = 676
```

Nous verrons plus en détail les passages de paramètres au paragraphe 2.25, page 48.

2.7 Déclaration et définition

On fait la distinction en C++ entre:

- la **déclaration** d'un identificateur, qui fait qu'on connaît ses **propriétés externes** et qu'on peut s'en servir dans d'autres déclarations;

- la **définition** d'un identificateur, qui précises toutes ses **propriétés internes**.

Cette distinction est tout à fait analogue à la séparation entre `interface` et `implementation` telle qu'on la voit dans les unités de compilation (`unit`) en Pascal.

Tout identificateur utilisé dans un ensemble de fichiers constituant un programme **peut être déclaré plusieurs fois**, et **doit être défini exactement une fois**.

Bien que cela anticipe en partie sur le Programmation orientée objets en ce qui concerne les classes, précisons d'emblée qu'**une déclaration est en général aussi une définition, sauf si**:

- il s'agit d'une déclaration de fonction sans son corps;
- elle contient le mot clé **`extern`**, mais pas d'*initialiseur* de la forme '= ...' ni de corps de fonction;
- elle est la déclaration d'un membre statique d'une structure ou d'une classe, ce qui sera traité en détail dans le Programmation orientée objets;
- elle consiste en la déclaration du nom d'une structure ou d'une classe, notion traitée dans le Programmation orientée objets.

Ainsi, on peut écrire:

```
#include <stream.h>

extern int i;           // une première déclaration de 'i'

extern int i;           // une seconde déclaration de 'i'

int i = 3;              // la définition de 'i' avec un initialiseur


int une_fonction ();    // déclaration de 'une_fonction'

int une_fonction ()     // définition de 'une_fonction'
     {
     return 33;
     }


class ma_classe;        // déclaration de 'ma_classe'

class ma_classe         // définition de 'ma_classe'
     {
     public:
          int               un_champ_decisif;
          static float      un_membre_statique;
               // déclaration de 'ma_classe :: un_membre_statique'
     };


float ma_classe :: un_membre_statique = 3.14;
               // définition de 'ma_classe::un_membre_statique'
```

```
main ()
     {
     cout <<
          i << "\n" <<
          une_fonction() << "\n" <<
          ma_classe :: un_membre_statique << "\n";
     }
```

Ce programme fournit comme résultat:

```
3
33
3.14
```

On retrouve l'analogie avec d'autre langages mentionnés ci-dessus dans le fait que lorsqu'on crée un '*module*' en C++, on met dans le fichier '**.h**' les **déclarations**, alors que le fichier '**.cp**' contient les **définitions**, comme on va le voir dans l'exemple du paragraphe suivant.

2.8 Gestion de modules: fichiers '.h' et fichiers '.cp'

Pour gérer des modules, on sépare la *déclaration* des facilités qu'il fournit à ses 'clients' des *définitions* nécessaires à l'implantation de ces facilités. Cela se fait en plaçant les déclarations dans le **fichier d'interface** **mon_module.h** et les définitions dans le **fichier d'implantation** **mon_module.cp**. Le suffixe '.h' vient de l'anglais *header*, entête en français.

Soit l'exemple de la gestion de dates, au sens du calendrier. On peut placer dans le fichier **Dates.h** les déclarations suivantes:

```
#ifndef __Dates__
#define __Dates__

#include <Types.h>


// Les dates
// ---------

struct DATE             // Description d'une date
     {
     short     jour;
     short     mois;
     short     annee;
     };


extern DATE    une_date_globale;
                        // déclaration de 'une_date_globale'


void ecrire_date (const DATE & une_date);

Boolean date_inf (const DATE & date_1, const DATE & date_2);

Boolean date_inf_egale (const DATE & date_1, const DATE & date_2);
```

```
long diff_date (const DATE & date_1, const DATE & date_2);

Boolean date_valide (DATE une_date);

#endif __Dates__
```

où l'on voit l'emploi de l'**inclusion conditionnelle** en fonction de la variable `__Dates__`, qui est définie lors de la première inclusion de ce fichier d'interface.

Le paramètre '`const DATE & une_date`' de la fonction `ecrire_date` indique que `une_date` est une **référence sur une variable non-modifiable** (`const &`) du type `DATE`. Le paramètre '`DATE une_date`' de `date_valide` est, lui, passé **par valeur**. Nous verrons au paragraphe 2.25, page 48, les détails de ces passages de paramètres.

On peut en fait écrire '`const DATE& une_date`', en collant le `&` contre le type, pour indiquer visuellement que le type est en fait '`DATE&`'. C'est là une question de goût personnel!

Le fichier `Types.h` est inclus pour que l'on puisse utiliser le type `Boolean`.

On peut alors placer dans le fichier `Dates.cp` les définitions:

```
#include "Dates.h"
#include <stream.h>

//   Les dates
//   ---------

void ecrire_date (const DATE & une_date)
     {
     cout <<
          form (
               "(Jour: %d, Mois: %d, Annee: %d)",

               une_date.jour,
               une_date.mois,
               une_date.annee
               );
     }


DATE      une_date_globale;
                              // définition de 'une_date_globale'


Boolean date_inf (const DATE & date_1, const DATE & date_2)
     {
     return
          ( date_1.annee < date_2.annee )
               ? true
               :

               ( date_1.annee > date_2.annee )
                    ? false
                    :
```

```
                    ( date_1.mois < date_2.mois )
                         ? true
                         :

                         ( date_1.mois > date_2.mois )
                              ? false
                              :
                              ( date_1.jour < date_2.jour )
          ;
     }


Boolean date_inf_egale (const DATE & date_1, const DATE & date_2)
     {
     return
          ( date_1.annee < date_2.annee )
               ? true
               :

               ( date_1.annee > date_2.annee )
                    ? false
                    :

                    ( date_1.mois < date_2.mois )
                         ? true
                         :

                         ( date_1.mois > date_2.mois )
                              ? false
                              :
                              ( date_1.jour <= date_2.jour )
          ;
     }


long diff_date (const DATE & date_1, const DATE & date_2)
     {
     return 0;          // à compléter par le lecteur!
     }


const int      jours_par_mois [12] =   // un tableau non exporté
     /* jan fev mar avr mai jun jul aou sep oct nov dec */
     {  31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31  };


Boolean date_valide(DATE une_date)
     {
     return
          ( une_date.mois >= 1 )
          &&
          ( une_date.mois <= 12 )
          &&
          ( une_date.jour <= jours_par_mois[une_date.mois - 1] )
          &&
          ( une_date.annee >= 1904 )
          &&
```

```
            ( une_date.annee <= 2039 )
            ;
      }
```

On remarque dans ce fichier '.cp' **l'inclusion du '.h' correspondant**, l'**expression conditionnelle** 'condition ? valeur_si_vrai : valeur_si_faux', le tableau contenant les nombres de jours par mois, et l'opérateur '*et avec court-circuit*' &&. Tous ces aspects du langage sont traités plus loin.

Il est ensuite possible d'importer ce 'module' dans un autre fichier Main.cp contenant par exemple:

```
#include "Dates.h"
#include <stream.h>


main ()
     {
     DATE      date_debut    = {10, 01, 1922};
                                  // liste d'initialiseurs
     DATE      date_fin      = {20, 03, 1930};

     une_date_globale.jour   = 1;
     une_date_globale.mois   = 12;
     une_date_globale.annee  = 1800;

     if (date_inf (date_debut, date_fin))
          ecrire_date (date_debut);
     else
          ecrire_date (date_fin);
     cout << "\n";

     ecrire_date (une_date_globale);
     if (! date_valide (une_date_globale))
          cout << " invalide";
     else
          cout << " valide";
     cout << "\n";
     }
```

Dans l'exemple ci-dessus, on déclare trois variables du type DATE, initialisées, que l'on passe ensuite en paramètre par référence et par valeur aux fonctions du module **Dates**, selon les cas. Cet exemple illustre aussi l'emploi de la fonction de *négation* '!'. Les listes d'initialiseurs pour les membres d'une structure sont présentés au paragraphe 2.14, page 24.

Pour que les modifications de Dates.h et Dates.cp ne provoquent respectivement que les re-compilations strictement nécessaires, il faut décrire les **dépendances entres fichiers** de manière adéquate. Dans le cas de MPW C++, cela peut être fait dans le fichier Dates.make qu'on soumet à l'outil **make**, et contenant:

```
OBJECTS = ∂
          'WORK:Cours_C++:.h & .cp:Dates.cp.o' ∂
          'WORK:Cours_C++:.h & .cp:Main.cp.o'

Dates ff Dates.make {OBJECTS}
     Link -d -c 'MPS ' -t MPST ∂
```

```
            {OBJECTS} ∂
            "{CLibraries}"CSANELib.o ∂
            "{CLibraries}"Math.o ∂
            "{CLibraries}"CplusLib.o ∂
            #"{CLibraries}"Complex.o ∂
            "{CLibraries}"StdCLib.o ∂
            "{Libraries}"Stubs.o ∂
            "{Libraries}"Runtime.o ∂
            "{Libraries}"Interface.o ∂
            "{Libraries}"ToolLibs.o ∂
            -o Dates

'WORK:Cours_C++:.h & .cp:Dates.cp.o' ƒ ∂
                  Dates.make ∂
                  'WORK:Cours_C++:.h & .cp:Dates.h' ∂
                  'WORK:Cours_C++:.h & .cp:Dates.cp'
      CPlus  'WORK:Cours_C++:.h & .cp:Dates.cp'

'WORK:Cours_C++:.h & .cp:Main.cp.o' ƒ ∂
                  Dates.make ∂
                  'WORK:Cours_C++:.h & .cp:Dates.h' ∂
                  'WORK:Cours_C++:.h & .cp:Main.cp'
      CPlus  'WORK:Cours_C++:.h & .cp:Main.cp'
```

Le lecteur est renvoyé au support de cours MPW du même auteur pour la manière de placer un module comme `Dates` dans une **librairie** MPW.

☞ **Ne pas placer un fichier '.h' dans une commande de compilation**: un tel fichier d'interface n'est destiné *qu'à être inclus* dans un autre fichier, ce qui provoquera bien sûr sa compilation tout de même.

☞ Par contre, ne pas oublier d'**en faire dépendre tous les fichers qui l'incluent** par **`#include`** dans la description pour *make*, car ils doivent être recompilés si cet interface vient à être modifié.

Le programme ci-dessus produit comme résultat:

```
(Jour: 10, Mois: 1, Annee: 1922)
(Jour: 1, Mois: 12, Annee: 1800) invalide
```

2.9 Portée statique des identificateurs

La portée statique d'un identificateur (*lexical scope* en anglais) est la partie du texte source en cours de compilation dans laquelle **cet identificateur peut être utilisé** par des déclarations ou des instructions.

Il existe en C++ quatre genres de portée statique ou textuelle des identificateurs. Là encore, nous anticipons un peu sur le Programmation orientée objets en ce qui concerne les classes, de manière à regrouper tous les cas dans ce paragraphe:

- **fichier**:

  un identificateur déclaré en dehors de tout bloc, fonction ou classe est utilisable *depuis le point de sa déclaration jusqu'à la fin du fichier* ou il est *déclaré*. Un tel identificateur est dit **global**;
- **bloc local**:

  un identificateur déclaré dans un bloc, fonction ou classe est utilisable *depuis le point de sa déclaration jusqu'à la fin du bloc* ou il est déclaré, *à l'exception des blocs imbriqués où il fait l'objet d'une re-déclaration*. Ceci est le cas des paramètres de fonction, des identificateurs déclarés localement à une fonction ou dans un *bloc interne* imbriqué à n'importe quel niveau dans une fonction;
- **fonction**:

  un identificateur d'étiquette déclaré dans une fonction est utilisable *depuis le début du texte de la fonction* où il est déclaré *jusqu'à la fin du texte de cette fonction*. Seuls les identificateurs d'*étiquette* ont cette propriété.
- **classe**:

  les identificateurs des membres d'une classe sont utilisables dans le corps des fonctions membres de cette classe, après l'opérateur '`.`' appliqué à une instance de cette classe, après l'opérateur `->` appliqué à un pointeur sur une instance de cette classe, ou après l'opérateur d'accès à un niveau de déclaration '`::`' appliqué a l'identificateur de cette classe ou à une classe dérivée de cette dernière. Tout cela sera détaillé dans le Programmation orientée objets.

  Les identificateurs de classe, d'énumération et de type synonyme (introduits par `typedef`) déclarés dans une classe sont considérés comme étant déclarés au même niveau que la classe elle-même.

  Les identificateurs de *fonctions amies* ('`friend` functions' en anglais) d'une classe sont aussi considérés comme déclarés au même niveau que la classe elle-même.

☞ Le **point de déclaration** d'un identificateur est placé **juste après sa déclaration complète**, et **avant un initialiseur éventuel**.

Lors d'*imbrications de blocs* comme pour accéder aux membres statiques et fonctions membres statiques d'une classe, on dispose de l'**opérateur d'accès à un niveau de déclaration** '`::`', qui sera encore rencontré au Programmation orientée objets. Qu'il suffise de savoir que le mot clé `class` pourrait être simplement remplacé ici par `struct`.

Le programme suivant illustre ces différentes portées statiques et l'opérateur '`::`':

```
// Portées statiques et l'opérateur '::' en C++

#include <stream.h>
#include <types.h>
```

```
float     i = 3.141592;


class une_classe{
     public:
          enum jour {lundi, mardi, boff};
          static Boolean     i;
          static char *      une_fonction_membre ();
     }; // fin de "une_classe"


Boolean une_classe :: i = true;


char * une_classe :: une_fonction_membre ()
     { return form ("%d", i); }


Boolean fausse ()
     { return false; }


main ()
     {
     for( int i = 1; i <= 4; ++i)
          { }

     if (fausse ())
          goto ETIQUETTE;    // elle est déclarée plus loin!


     cout <<
          "i global = " <<
          form ("%f", ::i) << "\n";

     cout <<
          "i de 'une_classe' = " <<
          form ("%d", une_classe::i) << "\n";

     cout <<
          "i local = " <<
          form ("%d", i) << "\n";

     cout <<
          "une_classe :: une_fonction_membre () = " <<
          une_classe :: une_fonction_membre () << "\n";


     cout <<
          "une_classe :: mardi = " <<
          form("%d", une_classe :: mardi) << "\n";


     if (3 < 5)
          {
          // Voici un bloc interne à la fonction 'main'
          float i = 5.3;     // masque le 'int i' de 'main'
          cout << "i du bloc interne = " << i << "\n";
```

```
        }
    //  Ici, 'i' du bloc interne n'existe plus


ETIQUETTE:
    cout << "C'est fini\n";
    }
```

Ce programme fournit comme résultat:

```
i global = 3.141592
i de 'une_classe' = 1
i local = 5
une_classe :: une_fonction_membre () = 1
une_classe :: mardi = 1
i du bloc interne = 5.3
C'est fini
```

On a employé la notation `::i` pour accéder à la variable `i` **globale**, alors que `i` seul désigne la variable **locale** à la fonction `main`. Remarquons que le `i` dans le corps de la fonction membre `une_classe::une_fonction_membre` est en fait la *variable de classe* `une_classe::i`.

Une fonction membre est lexicalement imbriquée dans sa classe, même si sa définition est placée en dehors de celle-ci, ainsi qu'on le verra au Programmation orientée objets.

Remarquons que la déclaration de `i` dans la fonction `main`, bien que **locale** à l'instruction `for`, est en fait déclarée *au même niveau que cette instruction*, et peut donc être utilisée après l'instruction `for`.

Il en va tout autrement de l'identificateur `j` déclaré dans le bloc interne contrôlé par l'instruction `for` suivante, dont la visibilité est restreinte à ce bloc:

```
for( int i = 1; i <= 4; ++i )
    {
    int      j = 8;
    }
//  "j" n'est pas accessible ici
```

Notons encore que certaines variables sont **anonymes**, et qu'elle n'ont pas de portée statique. Nous en verrons des exemples lors du passage de *paramètres par référence* et au Programmation orientée objets.

2.10 Portée dynamique des variables

Il existe trois classes de variables selon leur portée dynamique ou durée de vie, qui va de leur *naissance* à leur *mort*:

- une variable **statique** a une durée de vie égale à celle du programme en cours d'exécution: elle naît au début de l'exécution de la fonction `main`, et meurt à la fin de l'exécution de cette dernière.

Le nom 'statique' vient de ce qu'une telle variable est allouée statiquement, une fois pour toutes, pour toute la durée du programme. C'est la cas de toutes les **variables globales** au sens défini au paragraphe précédent.

Toutes les variables statiques sont allouées en un endroit déterminé au chargement du programme.

Nous verrons au paragraphe 2.12, page 22, un exemple de variable *locale* statique;

- une variable **automatique** a une durée de vie égale à l'activation de la fonction ou du bloc où elle est déclarée: elle naît lorqu'on entre dans cette activation de la fonction, et meurt lorqu'on sort de cette activation. C'est cette naissance et cette mort automatique avec l'appel et le retour d'une fonction qui leur donne leur nom aux variables automatiques.

  C'est le cas des **paramètres formels** des fonctions, des **variables locales** aux fonctions et aux blocs imbriqués dans les fonctions.

  Toutes les variables automatiques sont gérées dans une **pile** ('*stack*' en anglais), puisque celles qui sont nées le plus recemment sont celles qui meurent en premier;
- une variable **dynamique** naît par un appel à l'opérateur `new` et meurt par un appel à l'opérateur `delete`. Le nom 'dynamique' vient de ce que ces variables sont **crées par le programme à l'exécution**, dynamiquement en fonction des ses besoins.

  Les variables dynamiques sont gérées dans un **tas de mémoire** ('*heap*' en anglais), puisqu'elles peuvent survivre à leur créateur, et qu'un *pile* n'est donc pas utilisable pour les allouer.

  Nous verrons au paragraphe 2.15, page 26, la gestion de l'allocation dynamique de la mémoire.

Les variables **anonymes** mentionnées au paragraphe précédent ont une durée de vie dépendante de leur genre, ce qui sera traité au paragraphe 2.6, page 9.

2.11 Forme des déclarations

Une caractéristique de C++ est **les déclarations sont faites de manière descriptive**.

Ainsi:

```
int      * pointeur_sur_un_entier;
```

signifie que '`* pointeur_sur_un_entier`' est un `int`, soit que `pointeur_sur_un_entier` est un pointeur sur `int`.

De la même manière:

```
float     tableau_de_float [12];
```

indique que `tableau_de_float [un_entier_de_0_à_11]` est un `float`, donc que `tableau_de_float` est un tableau de `float`.

Le **cas des fonctions** est parfois un peu plus complexe. Ainsi:

```
int       * fonction_retournant_un_pointeur ();
```

déclare `fonction_retournant_un_pointeur` comme une fonction retournant un pointeur sur un `int`, tandis que:

```
int       (* pointeur_sur_fonction) ();
```

déclare `pointeur_sur_fonction` comme un pointeur sur une fonction sans paramètres retournant un `int`.

Les **pointeurs sur des membres d'une classe** se déclarent:

```
type_du_membre (la_classe :: * pointeur_sur_membre)
```

ce qui déclare `pointeur_sur_membre` comme un pointeur sur un membre du type `type_du_membre` de la clsse `la_classe`.

Les **pointeurs sur des fonctions membres d'une classe** se déclarent:

```
type_resultat (classe :: * pointeur_sur_fonction_membre)
     (paramètres)
```

ce qui déclare `pointeur_sur_fonction_membre` comme un pointeur sur une fonction membre de la classe `classe`, ayant certains paramètres éventuels et retournant une valeur du type `type_resultat` .

Une autre caractéristique de C++ est que **les déclarations sont des instructions particulières**, et qu'elle peuvent donc être placées au beau milieu du corps d'une fonction, **près de la portion de texte où elles sont utilisées**, comme dans:

```
#include <stream.h>
#include <types.h>


main ()
     {
     float          pi = 3.141592;
     cout << pi * pi << "\n";


     const int      max = 3;
     for (int i = 1; i <= max; ++i)
          cout << i * i << "\n";


     Boolean        drapeau = pi > 2;
     if (drapeau)
          cout << "drapeau est vrai\n";
     }
```

Ce programme produit comme résultat:

```
9.8696
1
4
9
drapeau est vrai
```

On notera au passage la déclaration de la **constante** `max` au moyen de:

```
const     int max = 3;
```

qui déclare `max` comme une *variable non-modifiable après son initialisation* (*'read-only'* en anglais). La valeur initiale d'une constante peut être n'importe quelle expression calculée à l'exécution du programme.

Nous verrons plus loin d'autres cas plus complexes d'emploi du mot clé `const`.

2.12 Variables locales statiques

Ainsi qu'il a été dit au paragraphe 2.10, page 19, une variable statique a une durée de vie égale à la durée d'exécution du programme lui-même.

Si une variable locale à une fonction est déclarée **static**, on la retrouve à chaque appel dans l'état où on l'avait laissée lors de l'appel précédent. La valeur initiale d'une telle variable est précisée à l'endoit de sa définition. Par contre, elle n'est visible que dans le corps de la fonction en question.

Une variable locale statique n'est donc rien d'autre qu'une **variable globale dont la visibilité est limitée au corps d'une seule fonction**. Comme toute les variables globales sont allouées statiquement, le mot clé `static` est employé dans la déclaration. Rappelons que le mot clé `static` désigne en C++ autant l'allocation mémoire qu'une visibilité restreinte.

Voici un exemple illustrant les **variables statiques locales à une fonction**:

```
#include <stream.h>

const float   pi = 3.141592;

float ma_function_a_memoire ()
     {
     static float       remanente = pi;
                             // celle-ci est statique


     float              automatique = remanente * 2;
                             // celle-ci est automatique


     remanente = automatique;

     return remanente;
     }


main ()
     {
     for (int i = 1; i <= 5; ++ i)
          cout << i << ": " << ma_function_a_memoire () << "\n";
     }
```

Ce programme produit comme résultat:

```
1: 6.28318
2: 12.5664
3: 25.1327
4: 50.2655
5: 100.531
```

car la spécification `static` fait que tous les appels à la fonction `ma_function_a_memoire` n'utilisent qu'une seule variable `remanente`. Par contre, une nouvelle variable `automatique` est créée à chaque appel de manière… automatique, ce qui fait que la valeur de `automatique` lors de l'appel précédent n'est pas conservée.

Sans le mot `static`, cette fonction aurait retourné invariablement la valeur `6.28318`, et le résultat serait:

```
1: 6.28318
2: 6.28318
3: 6.28318
4: 6.28318
5: 6.28318
```

Nous verrons au Programmation orientée objets que les *membres des classes* sont un autre moyen de disposer de variables à visibilité limitée mais qui survivent à un appel de fonction.

2.13 Types énumérés

On peut définir des **types entiers énumérés** comprenant des **valeurs nommées par des constantes**. Par défaut, les valeurs associées à ces constantes vont *de* `0` *à* `n-1`, où `n` est le nombre des constantes du types, comme dans:

```
enum couleur { rouge, vert, bleu }
```

où l'on définit le type `couleur` et les constantes `rouge`, `vert` et `bleu` avec les valeurs respectives `0`, `1` et `2`.

☞ Les constantes définies dans un type énuméré sont *de ce type*, qui est **distinct de tous les autres types entiers**.

☞ On peut convertir une constante énumérée comme `rouge` en un entier, mais la conversion inverse n'est pas possible:

```
couleur   c = 1;        // ILLEGAL
int       i = rouge;    // 'i' vaut initialement 0
```

On peut aussi **donner une valeur explicite à une constante énumérée**, ce qui fait que toutes celles qui suivent sont par défaut associées aux valeurs qui la suivent, comme dans:

```
enum couleur { rouge, vert = 10, bleu, jaune = 30, violet }
```

où l'on a les associations de valeurs suivantes:

```
rouge     : 0
vert      : 10
bleu      : 11
jaune     : 30
violet    : 31
```

2.14 Structures

Une structure est une **agrégation de plusieurs composants** appelés **membres** en C++, chacun pouvant être d'un type différent. Ainsi:

```
struct DATE             // Description d'une date
     {
     short     jour;
     short     mois;
     short     annee;
     };
```

spécifie que toute valeur du type `DATE` contient trois membres nommés `jour`, `mois` et `annee`, chacun étant du type `short`.

Si l'on déclare `une_date_globale` au moyen de:

```
DATE      une_date_globale;
```

on peut accéder à ses membres par:

```
une_date_globale.jour
une_date_globale.mois
une_date_globale.annee
```

respectivement.

Il est possible d'*affecter* une valeur d'un type structure, comme de la passer en *paramètre par valeur* et en *paramètre par référence* à une fonction. Le détail de ce qui se passe lors du passage par valeur sera traité au Aspects avancés du langage.

C++ offre une **notation syntaxique pour initialiser** des variables de types structure au moyen d'une **liste d'initialiseurs** des membres, dans l'ordre de leur déclaration, placés entre des accolades { et }, et séparés par des virgules, comme dans:

```
DATE      date_debut     = {10, 1, 1922};
```

☞ **S'il manque des valeurs** entre les accolades, les champs correspondants sont **remplis de zéros**, indépendamment de leur type.

☞ **On peut imbriquer récursivement des listes d'initialiseurs** dans le cas où certains membres sont eux-mêmes d'un type structure ou tableau. Ces derniers se prêtent aussi à ce mécanisme, comme on le verra au paragraphe 2.17, page 30.

☞ Attention que si l'on écrit:

```
DATE      date_debut     = {10, 09, 1922};
```

il y a une **faute de syntaxe**, car le chiffre zéro débute une constante octale, et `9` n'est pas un chiffre octal!

Voici un programme illustrant l'emploi de structures:

```
#include <stream.h>

// un type structure:
struct boite
     {
     int           l_entier;
     float         le_flottant;
     };

boite creer_une_boite ()                    // retourne une structure!
     {
     boite    res = { 17, 19.9 };
     return res;
     }

void ecrire_boite_par_valeur ( boite la_boite )
     {
     cout <<
          form (
               "ecrire_boite_par_valeur (%4d, %7.4f)\n",
               la_boite.l_entier, la_boite.le_flottant
               );
     }

void ecrire_boite_par_reference ( boite & la_boite )
     {
     cout <<
          form (
               "ecrire_boite_par_reference (%4d, %7.4f)\n",
               la_boite.l_entier, la_boite.le_flottant
               );
     }

main ()
     {
     boite    ma_boite;

     cout << "contenu initial de 'ma_boite':\n";
     ecrire_boite_par_valeur (ma_boite);

     ma_boite = creer_une_boite ();

     cout << "contenu final de 'ma_boite':\n";
     ecrire_boite_par_valeur (ma_boite);
     ecrire_boite_par_reference (ma_boite);
     }
```

La fonction `form` sert à mettre en forme des chaînes de caractères. Elle est présentée au paragraphe 2.20, page 37.

Le programme ci-dessus produit comme résultat:

```
contenu initial de 'ma_boite':
ecrire_boite_par_valeur (5502564,  0.0000)
contenu final de 'ma_boite':
ecrire_boite_par_valeur (  17, 19.9000)
ecrire_boite_par_reference (  17, 19.9000)
```

On voit que *la valeur initiale de* `ma_boite` *est quelconque*. Il est à remarquer que lors d'un passage d'une structure **par référence**, comme pour la procédure `ecrire_boite_par_reference`, l'**accès aux membres de la structure référencée** s'écrit **de la même manière que lorsqu'il y a passage par valeur**, ce qui est une différence notable avec C. C++ est dans ce domaine similaire à Pascal.

Nous verrons au paragraphe 2.6, page 9, la notion de *référence* C++, et au Aspects avancés du langage le mécanisme détaillé du *passage par valeur d'une structure*, ainsi que le *retour d'une structure comme valeur d'une fonction*.

2.15 Pointeurs et allocation dynamique de variables

Un pointeur est un **accès indirect** à une variable d'un type quelconque, qui est dite la **variable pointée par ce pointeur**.

Pour déclarer simultanément *deux* variables de type pointeur sur `un_type`, on ne doit pas écrire:

mais:

```
un_type   * pointeur_1, pointeur_2;
     //   'pointeur_2' n'est pas un pointeur!
```

mais:

```
un_type   * pointeur_1, *pointeur_2;
```

On peut **affecter à une variable de type pointeur** sur `un_type` **l'adresse d'une variable** du type `un_type` ou **la valeur d'un autre pointeur** sur une variable du type `un_type`, comme dans:

```
un_type   une_variable;

pointeur_1 = & une_variable;

pointeur_2 = new un_type;

pointeur_1 = pointeur_2;
```

On accède à la variable pointée par `un_pointeur` au moyen de la notation:

```
* un_pointeur
```

Il est *parfois nécessaire de parenthéser cette notation* pour forcer l'association de l'opérateur * à `un_pointeur`, selon le contexte.

Lors de l'emploi d'un pointeur sur une structure , l'accès à un membre de la structure pointée s'écrit:

```
(* le_pointeur).le_membre
```

ou de manière équivalente:

```
le_pointeur -> le_membre
```

☞ **Une variable de type pointeur n'a pas de valeur initiale par défaut**: elle vaut initialement n'importe quelle configuration de bits qui se trouve là en mémoire!

☞ Il vaut mieux **ne pas utiliser de pointeurs en C++ pour effectuer des passages de paramètres par référence**, comme cela est nécessaire en C: il suffit d'employer pour cela les références, présentées au paragraphe 2.6, page 9.

Nous verrons au Aspects avancés du langage les **conversions de types**. Disons d'emblée qu'il existe un **type pointeur universel** '`void *`', en lequel tout pointeur peut être converti.

La **création des variables dynamiques** se fait par:

```
un_pointeur = new type_de_la_variable_pointée;
```

ou:

```
un_pointeur = new type_de_la_variable_pointée (initialiseurs);
```

auquel cas on peut indiquer la valeur initiale de la variable ainsi créée.

La **desctruction des variables dynamiques** allouées par l'opérateur `new` se fait par:

```
delete un_pointeur;
```

☞ Bien que la variable pointée par `un_pointeur` **n'existe plus** d'un point de vue sémantique après '`delete un_pointeur;`', il est possible d'y accédér néanmoins, ce qui peut réserver des **surprises désagréables**!

☞ Si on **ré-affecte un pointeur** qui se trouve être le seul accès à une variable créée dynamiquement par l'opérateur `new`, cette variable devient **inaccessible** et occupera la mémoire inutilement jusqu'à la fin de l'exécution du programme.

☞ Il est **fortement déconseillé de faire deux fois de suite** '`delete un_pointeur;`' sur le même `un_pointeur`!

Nous verrons au Aspects avancés du langage qu'il est possible de gérer soi-même de manière très fine la mémoire dynamique en **re-définissant les opérateurs `new` et `delete`**.

Voici un exemple illustrant l'emploi des pointeurs et l'allocation dynamique de variables:

```
#include "stream.h"

main ()
     {
     float     le_reel;
     float     * mon_pointeur, * ton_pointeur;


     cout << "adresse de le_reel = " << & le_reel << "\n";
     cout <<
          "valeur initiale de mon_pointeur = " <<
          mon_pointeur << "\n\n";


     mon_pointeur = & le_reel;
     (* mon_pointeur) = 68.5;
     cout <<
          "* mon_pointeur = " << * mon_pointeur <<
          ", adresse = " << mon_pointeur << "\n\n";


     ton_pointeur = mon_pointeur;
     (* ton_pointeur) = 29.3;
     cout <<
          "le_reel = " << le_reel << "\n" <<
          "* mon_pointeur = " << * mon_pointeur <<
          ", adresse = " << mon_pointeur << "\n\n";


     mon_pointeur = new float;
     cout <<
          "* mon_pointeur = " << * mon_pointeur <<
          ", adresse = " << mon_pointeur << "\n";


     ton_pointeur = new float (12.7);
     cout <<
          "* ton_pointeur = " << * ton_pointeur <<
          ", ton_pointeur = " << ton_pointeur << "\n\n";


     delete ton_pointeur;
     cout << "on vient de détruire '* ton_pointeur'\n";


     cout <<
          "* ton_pointeur = " << * ton_pointeur <<
          ", ton_pointeur = " << ton_pointeur << "\n";
     }
```

Il produit comme résultat:

```
adresse de le_reel = 0x3b63a0
valeur initiale de mon_pointeur = 0x3b6444
```

```
* mon_pointeur = 68.5, adresse = 0x3b63a0

le_reel = 29.3
* mon_pointeur = 29.3, adresse = 0x3b63a0

* mon_pointeur = 0, adresse = 0xfd824
* ton_pointeur = 12.7, ton_pointeur = 0xfd830

on vient de détruire '* ton_pointeur'
* ton_pointeur = 8.80984e+12, ton_pointeur = 0xfd830
```

On remarque l'opérateur de **prise d'adresse** `&` dans l'affectation:

```
mon_pointeur = & le_reel;
```

qui fait que `mon_pointeur` pointe ensuite sur `le_reel`.

Le fait d'avoir **plusieurs pointeurs sur une même variable** fait qu'on a ainsi des **alias**: on peut alors accéder par **plusieurs notations distinctes** à cette variable.

C'est ainsi que, dans cet exemple, on modifie la variable `le_reel` par:

```
(* mon_pointeur) = 68.5;
```

et qu'on relit la nouvelle valeur par:

```
* ton_pointeur
```

Il n'est pas nécessaire de parenthéser `* mon_pointeur` devant l'**affectation** '`=`': nous l'avons fait pour expliciter le fait que *c'est la variable pointée par qui est modifiée*. L'extrait ci-dessus aurait donc pu s'écrire:

```
* mon_pointeur = 68.5;
```

2.16 Arithmétique des pointeurs

Un pointeur est en C++ un entier particulier, représentant *une adresse en mémoire*. Il est donc possible de faire de l'**arithmétique des pointeurs**, par exemple pour ajouter un déplacement relatif à un pointeur donné ou pour mesurer la '*distance*' séparant deux adresses.

C++ fait l'arithmétique des pointeurs en tenant compte de la taille des éléments pointés, d'après la déclaration des pointeurs.

Ainsi, dans:

```
int       mon_entier;

int       * pointeur_1 = & mon_entier;

int       * pointeur_2 = pointeur_1 + 3;
```

`pointeur_2` pointe à une adresse située '**3 `int` plus loin que `pointeur_1`**', soit 12 octets si un `int` occupe 32 bits `(sizeof (int) == 4)`.

Nous verrons au paragraphe suivant et au paragraphe 2.19, page 34, des exemples d'arithmétique des pointeurs dans les cas des tableaux et des *chaînes de caractères*.

2.17 Tableaux

Un tableau est une **agrégation de plusieurs composants** appelés **éléments**, chacun étant *d'un même type donné*. On accède à ces éléments en **indexant le tableau** par des **valeurs entières** appelées **indices**. C++ permet de manipuler des tableaux à **plusieurs dimensions**.

On définit un tableau, dans le cas monodimensionnel, par:

```
type_des_elements  le_tableau [nombre_d_elements];
```

et, dans le cas multi-dimensionnel, par:

```
type_des_elements
     le_tableau [nombre_d_elements_1] … [nombre_d_elements_n];
```

On accède à un élément d'un tableau, dans le cas mono-dimensionnel, par:

```
le_tableau [l_indice]
```

et, dans le cas multi-dimensionnel, par:

```
le_tableau [l_indice_1] … [l_indice_n];
```

☞ **Les différents indices doivent être compris entre 0 et le nonbre d'éléments correspondant moins 1**.

☞ **Le code objet engendré par C++ ne vérifie pas les indices: c'est au programmeur qu'il incombe de vérifier que tous les indices qu'ils emploie soient dans le bon intervalle**.

Nous verrons toutefois au Aspects avancés du langage que l'on peut re-définir l'opérateur d'indexation '`[]`' pour éviter cet inconvénient *majeur*, dicté par la recherche de la même efficacité que celle des programmes écrits en C.

Soit l'exemple:

```
#include <stream.h>

const int     lignes    = 2;
const int     colonnes  = 3;


//   un type tableau bi-dimensionnel:
typedef int   tab_int [lignes] [colonnes];


void ecrire_tab_int ( tab_int le_tableau )
     {
```

```
     for (int i = 0; i < lignes; ++i)
          for (int j = 0; j < colonnes; ++j)
               cout <<
                    form (
                    "   element [%2d, %2d] = %3d\n",
                         i, j, le_tableau [i] [j] );
     }


main ()
     {
     tab_int   tab;              // un premier tableau

     tab_int   autre_tab =       // un autre tableau, initialisé
          {
               { 3, 6, 4 },
               { 9, 7, 2 }
          };


     for (int i = 0; i < lignes; ++ i)
          for (int j = 0; j < colonnes; ++ j)
               tab [i] [j] = i - j;


     cout << "contenu de 'tab':\n";
     ecrire_tab_int (tab);

     cout << "\n";

     cout << "contenu de 'autre_tab':\n";
     ecrire_tab_int (autre_tab);
     }
```

Pour le détail des arguments de la fonction `form`, on se reportera au paragraphe 2.20, page 37.

Le programme ci-dessus fournit comme résultat:

```
contenu de 'tab':
   element [ 0,  0] =   0
   element [ 0,  1] =  -1
   element [ 0,  2] =  -2
   element [ 1,  0] =   1
   element [ 1,  1] =   0
   element [ 1,  2] =  -1


contenu de 'autre_tab':
   element [ 0,  0] =   3
   element [ 0,  1] =   6
   element [ 0,  2] =   4
   element [ 1,  0] =   9
   element [ 1,  1] =   7
   element [ 1,  2] =   2
```

On y déclare un type **tableau bi-dimensionnel** `tab_int` formé d'éléments du type `int`. La fonction `main` déclare ensuite (et définit) deux tels tableaux, dont l'un est initialisé par la **liste d'initialiseurs**:

```
{
     { 3, 6, 4 },
     { 9, 7, 2 }
};
```

et l'autre voit ses éléments affectés un à un dans deux boucles **`for`** imbriquées.

Ces deux tableaux sont ensuite imprimés par la procédure `ecrire_tab_int`.

☞ **S'il manque des valeurs** entre les accolades d'une liste d'initialiseurs pour un tableau, les éléments correspondants sont **remplis de zéros**, indépendemment de leur type.

☞ **On peut imbriquer récursivement des listes d'initialiseurs** dans le cas où certains membres sont eux-mêmes d'un type structure ou tableau, comme on le voit sur cet exemple.

☞ Lorqu'on passe un **tableau en paramètre à une fonction**, c'est toujours l'**adresse du tableau** qui est passée. En fait, **un tableau n'est qu'un pointeur sur le premier élément** (**celui d'indice 0**).

☞ Si l'on déclare un tableau `tab` comme par exemple:

```
char      tab [12];
```

`tab` est en fait une constante de type '**`char *`**'. Il n'a donc pas de sens en C++ d'affecter une valeur à cet identificateur.

☞ Un tableau n'étant qu'un pointeur sur le premier élément, l'**accès par indiçage** aux éléments de ce tableau se fait par l'**arithmétique des pointeurs**, et la notation:

```
tableau [indice]
```

est toujours équivalente à:

```
* (tableau + indice)
```

Citons encore qu'il est possible de définir et d'initialiser un tableau par une **liste d'initialiseurs**, comme dans:

```
char * MotsCles [] =
     {
     "### L'element d'indice 0! ###",
     "char", "short", "int", "*", "&", "const", "unsigned"
     };
```

où `MotsCles` est un tableau de `8` valeurs du typ '`char *`', dont les indices vont de `0` à `7` inclus.

2.18 Tableaux dynamiques

Etant données les définitions:

```
typedef       float     * tab_float;
const int               taille = 18;
```

on peut **allouer dynamiquement des tableaux**, au moyen de:

```
tab_point tab = new float [taille];
```

et le **détruire** ensuite par:

```
delete [] tab; }
```

Voici un exemple illustrant cette possibilité:

```
//   Allocation dynamique d'un tableau

#include <stream.h>

typedef int   * tab_int;


tab_int creer_tableau (
     short la_taille, int la_valeur_initiale
     )
     {
     tab_int   aux = new int [la_taille];

     for (int i = 0; i < la_taille; ++ i)
          aux [i] = la_valeur_initiale;

     return aux;
     }


void ecrire_tab_int (
     char * le_nom, tab_int le_tableau, short la_taille
     )
     {
     for (int i = 0; i < la_taille; ++ i)
          cout <<
               form(
                    "   %s [%2d] = %3d\n",
                    le_nom, i, le_tableau [i] );
     }


void detruire_tableau (
     char * le_nom, tab_int le_tableau, short la_taille
     )
     {
     cout << form ("contenu de '%s':\n", le_nom);
     ecrire_tab_int (le_nom, le_tableau, la_taille);

     delete [] le_tableau;
     }
```

```
main ()
     {
     const short    taille = 4;

     tab_int        mon_beau_tableau = creer_tableau (taille, 150);

     for (int i = 0; i < taille; ++ i)
          mon_beau_tableau [i] += i;

     detruire_tableau ("mon_beau_tableau", mon_beau_tableau, taille);
     }
```

Ce programme produit les résultats suivants:

```
contenu de 'mon_beau_tableau':
   mon_beau_tableau [ 0] = 150
   mon_beau_tableau [ 1] = 151
   mon_beau_tableau [ 2] = 152
   mon_beau_tableau [ 3] = 153
```

☞ Il est possible avec l'opérateur **new** de spécifier un *initialiseur* entre parenthèses, comme dans:

```
int       * int_ptr = new int (317);
cout << * int_ptr;
```

On n'alloue dans ce cas **qu'une seule variable du type int dont la valeur initiale est 317, et non pas un tableau de 317 éléments de ce type!**

Le danger est ici qu'il est facile de confondre les deux forme syntaxiques du new, et que l'absence de test sur les bornes empêche de dépister l'accès à ces 316 éléments manquants!

☞ Bien qu'il soit possible de gérer l'allocation dynamique de tableaux multi-dimensionnels avec new, les contraintes du langage dans ce cas font qu'il est beaucoup plus avantageux de passer par la définition d'une **classe**, comme nous le montrerons au Programmation orientée objets.

2.19 Chaînes de caractères

Une chaîne de caractères C++ est **un tableau de char**, donc un '**char ***', qui n'est autre que **le pointeur sur le premier caractère de la chaîne**. Comme cela a été dit au paragraphe 2.3, page 5, toute chaîne est terminée par un *caractère de code nul* '\0'.

☞ Une chaîne étant un tableau, lorsqu'elle est **passée en paramètre** à une fonction, c'est en fait **l'adresse du premier caractère qui est passée en paramètre, par valeur ou par référence selon le cas!**

☞ La déclaration '**char * une_chaine;**' **n'alloue de la place que pour le pointeur, pas pour les caractères eux-mêmes**!

Une constante de chaîne comme:

```
"Bien le bonjour"
```

est allouée lors du chargement du programme exécutable en mémoire, dans la zone des *variables globales*. C'est bien entendu l'adresse de son premier caractère qui est manipulée par le code objet.

Pour **allouer de la place pour les caractères d'une chaîne**, on peut déclarer explicitement un **tableau de `char`** comme dans:

```
const int     taille = 256;
typedef       char     Str255 [taille];

Str255        une_chaine, une_autre_chaine;
                   //   alloue la place pour ces deux chaînes
```

On peut aussi allouer dynamiquement la chaîne par l'opérateur **`new`**:

```
char          * chaine_dynamique = new char [taille];
```

qui crée une zone dynamique pouvant recevoir `taille` caractères, initialisée avec des caractères de code nul `'\0'`.

Voici un exemple de gestion de chaîne illustrant ces différents points:

```
#include <stream.h>

int longueur_chaine (char * une_chaine)
     {
     char      * curseur = une_chaine;


     while ( * curseur != '\0' )
          ++ curseur;                 //   passe au caractère suivant


     return curseur - une_chaine;     //   arithmétique des pointeurs
     }


void ecrire_chaine (char *une_chaine)
     {
     while ( * une_chaine != '\0' )
          cout <<
               form ( "%c", * (une_chaine ++) );
                    // ecrit un caractère et passe au suivant
     cout << "\n";
     }
```

```
void copier_chaine (char *source, char *destination)
     {
     while (
          ( * (destination ++) = * (source ++) )
                         //   affecte un caractère et passe au suivant
               !=
          '\0'
                         //   sort de la boucle si fin de chaîne
          )
          ;              //   rien a faire de plus à cet endroit
     }


const int      taille = 256;
typedef        char      Str255 [taille];


main ()
     {
     Str255    chaine_statique;
     char      * chaine_dynamique = new char [taille];
     char      * pointeur;


     cout << "chaine_statique initiale:\n";
     ecrire_chaine (chaine_statique);


     cout << "chaine_dynamique initiale:\n";
     ecrire_chaine (chaine_dynamique);


     pointeur = "Bien le bonjour";
     cout << "pointeur:\n";
     ecrire_chaine (pointeur);


     copier_chaine ("Je vous y prends!", chaine_statique);
     cout << "chaine_statique finale:\n";
     ecrire_chaine (chaine_statique);
     }
```

Ce programme produit comme résultat:

```
chaine_statique initiale:
   chaine de longeur 6
   s‡@Äö

chaine_dynamique initiale:
   chaine de longeur 0

pointeur:
   chaine de longeur 15
   Bien le bonjour
chaine_statique finale:
   chaine de longeur 17
   Je vous y prends!
```

☞ **On pourrait écrire** par exemple `* destination++ = * source ++` au lieu de `* (destination ++) = * (source ++)` au vu des priorités définies par C++ pour les opérateurs. Nous avons préféré cette écriture *parenthésée* pour expliciter l'*association* des opérateurs.

La valeur initiale de `chaine_statique` est ce qu'on trouve à cet endroit en mémoire jusqu'au prochaine caractère à code nul '`\0`', ce qui explique le '`s‡@Äö`' dans les résultats.

Cet exemple mise sur l'aspect '**langage d'expressions**' de C++: certaines opérations comme l'affectation '`=`' retournent une valeur qui peut être utilisée de suite, *au vol*. Ainsi dans:

```
( *(destination ++) = *(source ++) )
                        // affecte un caractère et passe au suivant
    !=
'\0'
```

on recopie un caractère de la chaîne `source` dans la chaîne `destination`, et on compare le caractère en question avec celui dont le code est nul, pour tester la fin de la chaine `source`.

Le lecteur aura remarqué qu'en fait les pointeurs `source` et `destination` *parcourent les deux chaînes*, puisqu'ils changent de valeur à chaque itération dans la boucle: cela est en accord avec la **sémantique usuelle du passage de paramètre par valeur**: il s'agit d'une **variable locale à la fonction appelée, dont la valeur initiale est fournie par l'argument d'appel**.

Il aurait aussi été possible d'utiliser deux variables auxilaires analogues à `curseur` déclaré dans `longueur_chaine` : cela est laissé en exercice pour le lecteur.

2.20 Mise en forme de chaînes de caractères

Il est temps de préciser les détails de l'emploi de la fonction `form`, placée dans la librairie stream, et qui est similaire à la fonction `sprintf` de C.

Soit l'exemple:

```
void ecrire_tab_int ( tab_int le_tableau )
     {
     for (int i = 0; i < lignes; ++i)
          for (int j = 0; j < colonnes; ++j)
               cout <<
                    form (
                    "   element [%2d, %2d] = %3d\n",
                         i, j, le_tableau [i] [j] );
     }
```

On écrit sur le flot de sortie `cout` une chaîne de caractères dont on fournit le modèle, soit:

```
"   element [%2d, %2d] = %3d\n"
```

Dans modèle ('format' en anglais), on laisse des trous spécifiés par **%...**, qui seront occupés par les arguments suivant le modèle dans l'appel à `form`.

2.21 Unions

Une union est une **structure à variantes**, qui contient à chaque instant l'un des membres déclarés, à l'exclusion des autres. La taille d'une union est **la taille du plus grand de ses membres**.

Les unions ne sont **à conseiller qu'à très bas niveau**, où leur **manque de sécurité** peut être accepté. Pour des structures de données de haut niveau, les **classes** présentées au Programmation orientée objets leur sont **infiniment supérieures**, puisqu'elles permettent de gérer des *variantes* de manière propre et sûre.

Voici un exemple d'emploi d'unions:

```
#include <stream.h>

//   un type structure utilisant une union:
union variantes
     {
     int            l_entier;
     float          le_flottant;
     };

struct boite_a_variantes
     {
     char           * la_chaine;       //   la partie commune
     variantes      les_variantes;     //   la partie 'variantes'
     };

boite_a_variantes creer_une_boite ()
     {
     boite_a_variantes
          res =
               { "une belle boite_a_variantes", { 19.9 } };
     return res;
     }

void ecrire_boite_par_valeur (boite_a_variantes la_boite)
     {
     cout <<
          form (
               "ecrire_boite_par_valeur (%s, %4d, %7.4f)\n",
               la_boite.la_chaine,
               la_boite.les_variantes.l_entier,
               la_boite.les_variantes.le_flottant );
     }
```

```
void ecrire_boite_par_reference (boite_a_variantes & la_boite)
     {
     cout <<
          form (
               "ecrire_boite_par_reference (%s, %4d, %7.4f)\n",
               la_boite.la_chaine,
               la_boite.les_variantes.l_entier,
               la_boite.les_variantes.le_flottant );
     }

main ()
     {
     boite_a_variantes      ma_boite;

     cout << "contenu initial de 'ma_boite':\n";
     ecrire_boite_par_valeur (ma_boite);

     ma_boite = creer_une_boite();

     cout << "contenu final de 'ma_boite':\n";
     ecrire_boite_par_valeur (ma_boite);
     ecrire_boite_par_reference (ma_boite);
     }
```

Ce programme produit comme résultat:

```
contenu initial de 'ma_boite':
ecrire_boite_par_valeur (, 2733890,  0.0000)
contenu final de 'ma_boite':
ecrire_boite_par_valeur (une belle boite_a_variantes,   19,  0.0000)
ecrire_boite_par_reference (une belle boite_a_variantes,   19,  0.0000)
```

L'**accès aux champs de l'union** se fait par la *notation pointée* comme pour les structures, à l'image de:

```
la_boite.les_variantes.le_flottant
```

On voit sur cet exemple le **manque de sécurité** dans la gestion des variantes mentionné plus haut: **rien ne permet de savoir à un moment donné quelle est la variante courante stockée dans une union**. En fait, il n'y a pas d'exclusion mutuelle entre les différentes variantes que représentent les membres d'une union. Il nous est ainsi possible d'accéder à la fois à `l_entier` et à `le_flottant` dans cet exemple.

On pourrait ajouter *un membre supplémentaire qui gèrerait la variante courante*, par exemple en le déclarant d'un type énuméré. Comme **c'est au programmeur qu'il incombe toutefois de gérer ce champ de manière consistante**, la sécurité n'en serait pas meilleure pour autant.

Il est aussi possible d'utiliser une **union anonyme**, dont les champs sont alors considérés comme déclarés directement dans le niveau de déclaration contenant l'union. Soit l'exemple:

```
struct autre_boite_a_variantes
     {
     char          * la_chaine;   //   partie commune

     union                        //   union anonyme (variantes)
          {
          int       l_entier;
          float     le_flottant;
          };
     };

autre_boite_a_variantes      une_grosse_boite;
```

On accède aux champs de `une_grosse_boite` par:

```
une_grosse_boite.la_chaine
une_grosse_boite.l_entier
une_grosse_boite.le_flottant
```

respectivement.

2.22 Instructions

Les instructions de C++ sont toujours terminées par un '`;`', qui fonctionne dans ce langage comme terminateur d'instruction.

Le groupement de plusieurs instructions en une seule **instruction composée** se fait **à l'aide d'accollades `{` et `}`** pour former ce qui est en fait un **bloc interne**, qui peut contenir des déclarations si on le désire. Un tel bloc interne n'est **jamais suivi de** '`;`'.

C++ offre la panoplie usuelle d'intructions de contrôle. **Sa sémantique stipule que, dans les instructions `if`, `while`, `do` et `for`:**

- **toute condition doit être une expression entière ou de type pointeur** (ce qui revient donc au même)**;**
- **une condition doit toujours être placée entre parenthèses;**
- des **conversions de types classes**, que nous verrons au Programmation orientée objets, en entiers ou en pointeurs sont **admises** dans une condition;
- **une expression de valeur non-nulle est vraie, tandis qu'une expression de valeur nulle est fausse.**

Une **condition** portant simplement sur la **valeur d'un pointeur** est **vraie si ce pointeur est non-nul**, et **fausse sinon**.

L'instruction `if` a l'une des deux formes:

```
if (une_condition)
     une_instruction_ou_un_bloc_interne
```

```
if (une_condition)
     une_instruction_ou_un_bloc_interne
else
     une_instruction_ou_un_bloc_interne
```

Dans le cas de `if` imbriqués, la résolution du `else` se fait de la manière usuelle, soit qu'un `else` correspond toujours au `if` le plus récemment rencontré avant lui dans le même bloc.

L'instruction **`while`** a la forme:

```
while (une_condition)
     une_instruction_ou_un_bloc_interne
```

On exécute l'instruction qu'elle contrôle tant que la condition `une_condition` est vraie, donc *différente de 0*. La condition est ré-évaluée à chaque itération, et **il se peut que l'instruction contrôlée par le `while` ne soit jamais exécutée**: c'est ce qui se passe si la condition est initialement égale à `0`. L'instruction `while` de C++ est donc tout à fait analogue au `while` de Pascal.

L'instruction **`do`** a la forme:

```
do
     une_instruction_ou_un_bloc_interne
while (une_condition);
```

On exécute l'instruction qu'elle contrôle jusqu'à ce que la condition `une_condition` soit vraie, donc *différente de 0*. La condition est ré-évaluée à chaque itération, et l'instruction contrôlée par le `while` est toujours exécutée **au moins une fois**. Cette instruction est donc analogue au `repeat` de Pascal.

L'instruction **`for`** a la forme:

```
for (initialisation; test_de_continuation; avancée)
     une_instruction_ou_un_bloc_interne
```

Elle est toujours sémantiquement équivalente à:

```
initialisation;

while (test_de_continuation)
     {
     une_instruction_ou_un_bloc_interne
     avancée;
     }
```

L'instruction contrôlée n'est donc pas exécutée si le `test_de_continuation` est initialement faux, soit égal à `0`.

L'initialisation peut déclarer une variable et l'initialiser, par exemple avec plusieurs expressions séparées par des virgules dont la dernière indique la valeur, comme dans:

```
int      somme;

for (int i = (somme = 0, 1); i <= 5; ++ i)
     somme += i * i;
```

```
cout << "Somme des carrés = " << somme << "\n";
```

Le code ci-dessus affiche la valeur `55` comme résultat.

Comme le type `Boolean` n'est qu'un entier, le contrôle de types de C++ ne voit pas de problème dans une construction comme:

```
for (int C = 1; C++; C <= 33)
     {
     //   … … … … …
     }
```

dans laquelle le test de fin de boucle '`C <= taille`' et l'action à prendre à chaque itération '`C++`' ont été lalencontreusement *permutés*, ce qui donne en fait une *boucle infinie*.

Il est possible de **sortir en cours de route d'une boucle** `while`, `do` ou `for` au moyen de l'instruction:

```
break;
```

qui sort de l'itération la plus imbriquée la contenant. L'exécution passe dans ce cas à ce qui suit cette instruction textuellement.

De manière duale, il est possible de **passer directement à la prochaine itération** d'une boucle par l'instruction:

```
continue;
```

qui fait continuer l'exécution à la condition contrôlant l'itération considérée.

L'instruction de discrimination de cas `switch` a la forme générale:

```
switch (une_expression)
     {
     case une_premiere_expression_constante:
          une_premiere_sequence_d_instructions
          break;
     case une_seconde_expression_constante:
          une_seconde_sequence_d_instructions
          break;
     default:                              // optionnel
          une_derniere_sequence_d_instructions
     }
```

Toutes les expressions constantes figurant dans les alternatives introduites par `case` doivent être distinctes. On peut grouper plusieurs expressions constantes devant une séquence d'instructions.

Notons que la partie '`default:`' est **optionelle**. L'instruction `break` indique que l'on doit sortir de l'instruction `switch` après avoir choisi l'alternative correspondante.

Bien que cela soit peu fréquent, **on peut omettre le `break`** dans une alternative si l'on désire **continuer l'exécution à l'alternative suivante**, comme dans:

```
switch (une_expression_caractere)
     {
     case 'a':
          traiter_a_minuscule_spécifiquement;
          //   PUIS

     case 'A':
          traiter_a_majuscule_spécifiquement;
          break;

     case 'b':
          // et ainsi de suite
… … … … …
     }
```

Dans le cas où `une_expression_caractere` est égale à `'a'`, on fait le travail pour le cas `'a'` puis celui pour le cas `'A'`, ensuite de quoi on sort du `switch`.

L'instruction de saut:

```
goto etiquette;
```

fait continuer l'exécution à l'étiquette `etiquette`, qui doit être définie *dans la fonction* où le `goto` apparaît. Une étiquette est un *identificateur* en C++, qui peut être utilisé avant d'être déclaré, comme on l'a vu au paragraphe 2.9, page 16.

2.23 Débranchements non-locaux

Il est possible en C++ de faire des **débranchements non-locaux**, surtout pour réaliser une **gestion des cas d'erreurs** rencontrés par une application. L'idée est de sauvegarder à un moment donné l'état de l'environnement d'exécution du programme et de restaurer cet état plus tard:

- `setjmp (le_buffer_de_saut)` a pour effet de **mémoriser**, dans la variable `le_buffer_de_saut`, **l'état de l'environnement d'exécution**, soit la hauteur de la pile et les valeurs des registres. L'appel '*normal*' à cette fonction retourne `0`;
- `longjmp (le_buffer_de_saut, une_valeur)` restaure l'état de l'environnement d'exécution à celui qui a été mémorisé lors du dernier `setjmp (le_buffer_de_saut)`. De plus, cet appel provoque le retour fictif de l'appel `setjmp (le_buffer_de_saut)` en question avec comme valeur retournée `une_valeur`;
- en employant pour `longjmp` une valeur non-nulle, on peut distinguer le retour 'normal' de `setjmp` d'un débranchement effectué par `longjmp`.

Voici un exemple illustrant les débranchements non-locaux:

```
#include <setjmp.h>
#include <stream.h>

jmp_buf       le_buffer_de_saut;          // doit être global

const int     calcul_sur_pair = 3;        // par exemple

void calcul (int n)
     {
     cout << "on entre dans 'calcul' avec 'n = " << n << "'\n";
     if (n % 2 == 0)
          {
          cout << "PROBLEME: 'n' est pair\n";
          longjmp (le_buffer_de_saut, calcul_sur_pair);
          }
     cout << "TOUT VA BIEN: 'n' est impair\n";
     }

main ()
     {
     int      resultat;

     //   on met en place ce qu'il faut pour récupérer
     //   le 'longjmp' éventuel
     if ( (resultat = setjmp (le_buffer_de_saut)) != 0 )
          {
          cout <<
               "retour de 'setjmp' avec 'longjmp', valeur = " <<
               resultat << "\n";
          goto sortie;
          }

     cout << "on y va avec calcul(23)\n";
     calcul (23);
     cout << "\n";

     cout << "on y va avec calcul(12)\n";
     calcul (12);
     cout << "\n";

     cout << "on a bien terminé les deux appels à 'calcul'\n";

sortie:
     cout << "\nc'est fini!\n";
     }
```

Ce programme fournit comme résultat:

```
on y va avec calcul(23)
on entre dans 'calcul' avec 'n = 23'
TOUT VA BIEN: 'n' est impair


on y va avec calcul(12)
on entre dans 'calcul' avec 'n = 12'
PROBLEME: 'n' est pair
retour de 'setjmp' avec 'longjmp', valeur = 3

c'est fini!
```

L'appel '*normal*' à `setjmp` retourne `0`, ce qui fait que l'on continue ensuite l'exécution avec les différents appels à `calcul`.

Dans le cas où calcul reçoit un argument pair, l'appel à `longjmp` provoque le retour fictif de l'appel `setjmp (le_buffer_de_saut)`, avec `calcul_sur_pair` comme valeur retournée. Comme cette valeur n'est pas nulle, ce retour fictif conduit dans notre exemple à sauter à l'étiquette `sortie`.

☞ Il faut que **`setjmp (le_buffer_de_saut)` ait été exécuté avant `longjmp (le_buffer_de_saut, une_valeur)`**, sans quoi l'effet peut être imprévisible.

☞ Si par inadvertance on faisait `longjmp (le_buffer_de_saut, 0)`, la valeur retournée fictivement par `setjmp` serait en fait `1`.

La gestion des exceptions proposée par C++ 3.0 est une généralisation à haut niveau des débranchements non-locaux, qui peut d'ailleurs être implantée au moyen de `setjmp` et `longjmp`.

2.24 Opérateurs courants

Rappelons qu'il n'y a pas en C++ de type booléen en tant que tel: tous les opérateurs logiques produisent les valeurs `0` et `1` pour 'faux' et 'vrai', respectivement.

Sans entrer dans le détail de tous les opérateurs du langage, citons simplement:

- les **opérateurs dyadiques de comparaison `==`, `!=`, `<`, `<=`, `>` et `>=`**, retournant `0` ou `1` pour faux et vrai, respectivement;
- les **opérateurs monadiques auto-incrémentant `++` et son jumeau auto-décrémentant `--`**, chacun pouvant être *préfixé* comme *postfixé*. Ainsi l'expression:

  ```
  ++ une_variable
  ```

  est sémantiquement équivalente à:

  ```
  (
  une_variable += 1,
  une_variable
  )
  ```

tandis que:

```
une_variable ++
```

est équivalente à un appel à la fonction:

```
inline anonyme ()
     {
     type_de_une_variable    ancienne_valeur =
                                  une_variable;
     une_variable += 1;
     return ancienne_valeur;
     }
```

- les opérateurs arithmétiques usuels `+`, `-`, `*`, et `/`. Dans le cas de `*` et `/`, les conversions usuelles sont appliquées aux opérandes et déterminent le type du résultat. Ainsi la **division entière** s'écrit simplement `/`, et:

```
cout << 5 / 2 << "  " << 5 / 2.0 << "\n";
```

produit comme résultat:

```
2  2.5
```

- l'opérateur **modulo** `%`, qui n'admet que *des opérandes entiers*. Si `b` est non-nul, on a toujours:

```
a == (a / b) * b + a % b
```

Si a et b sont non-négatifs, `a % b` est non-négatif, sinon son signe dépend de l'implantation. En pratique, il est souvent du même signe que `a`.

- les **opérateurs logiques** `!`, `&&` et `||`. Le premier est la *négation logique*, tandis que les deux derniers sont les opérateurs '*et logique avec court-circuit*' et '*ou logique avec court-circuit*', respectivement;

Le terme '**avec court-circuit**' signifie que **ces opérateurs n'évaluent leur second argument que si la valeur du premier ne permet pas de déterminer la valeur du résultat**. Ainsi, `&&` n'évalue le second argument que si le premier est non-nul, tandis que `||` ne l'évalue que si le premier est nul.

- l'opérateur conditionnel '`?:`' est tel que l'expression:

```
condition ? valeur_si_non_nul : valeur_si_nul
```

est équivalente à un appel à la fonction:

```
inline anonyme ()
     {
     if(condition)
          return valeur_si_non_nul;
     else
          return valeur_si_nul;
     }
```

- les **opérateurs bit à bit** (*bitwise operators* en anglais), comme ceux de décalages arithmétiques (*arithmétic shifts* en anglais) `<<` et `>>`, le '*et bit à bit*' `&`, le '*ou bit à bit*' `|`, le '*ou exclusif bit à bit*' `^` et la '*négation bit à bit*' `~`;
- les **opérateurs combinant une affectation avec une opération** arithmétique comme `+=`, `-=`, `/=`, `*=` et `%=`. Ainsi:

  ```
  une_variable += une_expression
  ```

  est équivalent à:

  ```
  une_variable = une_variable + une_expression
  ```

- l'opérateur '`,`' permet de **grouper des expressions** qui sont toutes évaluées, mais dont seule **la valeur de la dernière est retournée** comme valeur de l'expression composite. Ainsi:

  ```
  cout << (i = 5, i --, i * 2);
  ```

  écrit comme résultat:

  ```
  8
  ```

  et laisse la valeur `4` dans la variable `i`.
  Il est nécessaire de *parenthéser ce groupement* dans les contextes où la virgule joue un autre rôle syntaxique, comme lorsqu'il est lui-même un argument suivi d'une virgule dans un appel de fonction paramétrée.

Dans une condition logique, les comparaisons '`== 0`' et '`!= 0`' sont **superflues** lorsqu'elles constituent l'opérateur principal. Ainsi:

```
if (a == 0)
     if (b != 0)
          { // … … … … }
```

peut être écrit de manière équivalente:

```
if (! a)
     if (b)
          { // … … … … }
```

Il faut se méfier, surtout lorqu'on vient de langages dérivés d'Algol 60 comme Pascal, de **ne pas écricre par inadvertance**:

```
if (une_variable = une_expression)
```

car la condition ci-dessus **affecte** `une_expression` à `une_variable` et **retourne comme résultat la valeur affectée**. Le compilateur ATT fournit un avertissement dans ce cas, mais *pas dans celui où l'affectation est une sous-expression*, comme dans:

```
if ((une_variable = une_expression && (autre_chose))
```

Méfiance, méfiance!

☞ Rappelons que tous les opérateurs arithmétiques se prêtent à l'*arithmétique des pointeurs*, qui a été présentée au paragraphe 2.16, page 29.

2.25 Passages de paramètres

C++ propose deux modes de passages de paramètres:

- le **passage par référence**, caractérisé par le marqueur syntaxique **&**, analogue à celui de Pascal et Modula-2:

```
carre (int & n)
```

- le **passage par valeur**, dans tous les autres cas.

Rappelons que la définition et l'appel d'une **fonction sans paramètres** se fait en indiquant une liste d'arguments vide entre parenthèses, comme dans:

```
int f ()
    { … }

… f () …
```

Voici un exemple illustrant les différents modes de passage de paramètres, y compris le passage d'une fonction en paramètre:

```
#include <stream.h>
#include <math.h>

typedef long double     reel;

const reel epsilon      = 0.01;
const reel pi           = 3.141592;

void tabuler (
          reel          funct (reel r),
          reel          borne_inf,
          reel          born e_sup,
          reel          pas,
          char          * r,
          char          * funct_de_r,
          reel          & valeur_cumulee
          )

    {
    cout << form ("%5s %13s\n", r, funct_de_r);
    valeur_cumulee = 0;

    while (borne_inf <= borne_sup)
          {
          reel      valeur = funct (borne_inf); //   appel a "funct"
```

```
            cout << form("%7.3f  %7.3f\n", borne_inf, valeur);
            valeur_cumulee += valeur;
            borne_inf += pas;
            }
        } // tabuler

reel tangente (reel r)
        { return sin (r) / cos (r); }

main ()
        {
        reel      total_sinus, total_tangente;

        tabuler (
            sin, 0, pi / 2, 0.3, "X", "Sinus(x)", total_sinus
            );
        cout << form ("Total:   %7.3f\n", total_sinus);
        cout << "\n";
        tabuler (
            tangente, 0, pi / 2, 0.3, "X", "Tangente(x)", total_tangente
            );
        cout << form ("Total:   %7.3f\n", total_tangente);
        }
```

La procédure `tabuler` reçoit une **fonction en paramètre**: `funct` est une fonction d'un paramètre réel produisant un réel, et est donc déclarée de manière descriptive par:

```
reel funct (reel r)
```

Les arguments d'appels sont des *fonctions ayant ce même profil*, soit une fois `sin` qui est déclarée dans le fichier **`math.h`**, et l'autre fois la fonction `tangente` définie par le programmeur.

Les **chaînes de caractères** comme `"X"` et `"Sinus(x)"` sont passées **par valeur**, mais n'oublions par qu'une chaîne n'est qu'un pointeur sur son premier caractère: c'est donc ce pointeur qui est passé par valeur.

Le paramètre `valeur_cumulee`, passé *par référence*, permet de manipuler la variable fournie comme argument d'appel depuis le corps de la procédure `tabuler`).

Les résultats produits par ce programme sont:

```
    X      Sinus(x)
  0.000    0.000
  0.300    0.296
  0.600    0.565
  0.900    0.783
  1.200    0.932
  1.500    0.997
Total:     3.573


    X   Tangente(x)
  0.000    0.000
```

```
  0.300    0.309
  0.600    0.684
  0.900    1.260
  1.200    2.572
  1.500   14.101
Total:    18.927
```

Il est possible de passer un **paramètre par valeur constante**, ce qui fait qu'on ne peut le modifier dans le corps de la fonction. Ainsi, l'exemple suivant est rejeté par le compilateur avec le message apparaissant en commentaire:

```
int f (const int n)
     {
     ++ n;
     // ERREUR: increment of  constant n
     }
```

C++ permet aussi le **passage de paramètres par référence de manière non-modifiable**. La référence étant un synonyme n'est bien sûr pas modifiable, mais **la variable référencée n'est pas modifiable non plus dans ce cas**. Voici un exemple illustrant cela:

```
long carre (const long & n)
     {
     ++ n;
     // ERREUR: increment of  constant
     return n*n;
     }
```

On peut spécifier des **valeurs par défaut pour les arguments** des fonctions. Cela se fait en faisant suivre les paramètres formels corresondants par '`= une_expression`'. Tous les paramètres pour lesquels on indique une valeur par défaut doivent être regroupés en fin de liste des paramètres.

On voit cela à l'œuvre dans l'exemple suivant:

```
#include <stream.h>

int       entier_global = 2;


int fonction_globale (int n)
     { return n * n; }


int fonction (
          int  parm_1 = 3,
          int  parm_2 = fonction_globale (entier_global)
          )
     { return parm_1 * parm_2; }


main ()
     {
     cout <<
          form (
               "%d, %d, %d\n",
```

```
                fonction (), fonction (5), fonction (7, 9)
                );
        }
```

qui produit comme résultat:

```
12, 20, 63
```

La valeur par défaut d'un paramètre formel pour lequel aucun argument d'appel n'est fourni est **(ré-)évaluée à chaque appel**. Cette évaluation se fait **dans le contexte où la valeur par défaut a été spécifiée**, et non dans le contexte de l'appel.

Il est possible de spécifier des **paramètres facultatifs** à l'aide de la notation **...**: les paramètres qui précèdent doivent être fournis à chaque appel, tandis qu'on peut en fournir d'autres à volonté. L'exemple de `printf`, déclarée dans `stdio.h`, illustre cette possibilité:

```
printf (char * format_d_impression, …)
```

puisqu'on peut l'appeler par:

```
printf ("blark")
```

aussi bien que par:

```
printf ("prix %d", 3*19)
```

Ceci n'est en fait utile que pour des fonctions de librairies écrites en d'autres langages, puisqu'il n'existe pas de notation C++ pour accéder aux arguments optionnels s'ils sont effectivement passés dans un appel.

Les arguments d'appel sont **convertis s'il le faut au type des paramètres formels**, lesquels sont en fait **initialisés par la valeur de l'argument d'appel**. Nous verrons au Aspects avancés du langage que l'on peut définir soi-même des fonctions de conversion en complément aux conversions pré-définies par C++.

C++ permet l'écriture de **fonctions récursives** sans restriction.

En revanche, on ne peut **pas imbriquer une fonction textuellement dans une autre** comme on le fait en Pascal.

2.26 Surcharge sémantique

Il est possible en C++ de définir **plusieurs versions d'une même fonction**, pour autant qu'**elles diffèrent par leur profil**. Cette **surcharge sémantique** (*overloading* en anglais) permet d'écrire du code plus direct, par exemple en permettant d'écrire:

```
3 + 5
```

comme:

```
une_matrice + une_autre_matrice
```

ou:

```
une_matrice + 3.5
```

qui pourrait avoir pour effet d'ajouter `3.5` à chaque élément de la matrice `une_matrice`.

La détermination de **quelle version appeler en fonction des arguments d'appel** se fait en cherchant celle qui offre la **meilleure adéquation** (*best match* en anglais). Cette fonction, si elle existe, est une de celles qui offre la meilleure adéquation sur chacun des arguments considérées isolément. Si plusieurs telles fonctions existent, il y a **ambiguïté** dans la surcharge, et le programme est sémantiquement refusé à la compilation.

Le **profil** utilisé en C++ pour déterminer quelle fonction appeler en cas de surcharge sémantique ne concerne **que les arguments**, **et non le type de la valeur retournée par ces fonctions**.

Une fonction est **adéquate pour un argument** s'il est possible de convertir l'argument d'appel en le paramètre formel correspondant de cette fonction. Pour cela, on considère **dans l'ordre** suivant les cas d'adéquation:

- parfaite;
- après **promotion** d'une valeur entière à un autre type entier;
- après conversion standard;
- après conversion nécessitant la création d'une *variable anonyme temporaire*, comme on l'a vu au paragraphe 2.6, page 9;
- après une conversion définie par le programme;
- avec un paramètre optionnel (...);

En pratique, on doit se préoccuper de ce problème surtout pour les types entiers et flottants. De toute manière, *toute ambiguïté invalide le programme* et est donc signalée à la compilation.

Voici un exemple simple illustrant la surcharge sémantique:

```
#include <stream.h>

void traiter_chaine (char * la_chaine)
     {
     cout << form ("%10s | %10s\n", la_chaine, la_chaine);
     };


void traiter_chaine (int i)
     {
     traiter_chaine (form ("%d", i));
                                        // PAS UN APPEL RECURSIF!
     };


main ()
     {
     traiter_chaine (3245);
     traiter_chaine ("3.9");
```

```
}
```

Ce programme fournit comme résultat:

```
3245 |       3245
 3.9 |        3.9
```

Nous verrons des exemples plus significatifs de surcharges sémantiques dans les chapitres suivants.

Contrairement à ce que nous verrons dans le cas de la re-définition de fonctions membres dans des sous-classes au Programmation orientée objets, **toutes les versions surchargées d'une même fonction sont déclarées au même niveau**, dans le même bloc. Des fonctions de même nom déclarées dans des niveaux de déclarations différents sont simplement *distinctes*.

2.27 Flots d'entrées/sorties

C++ ne contient pas dans le langage lui-même de primitives de gestion d'entrées/sorties. C'est la librairie **<stream.h>** permet la gestion de flots d'entrées/sorties de manière abstraite, indépendamment du dispositif fournissant ou recevant effectivement les caractères. Les flots pré-définis sont:

cin	flot courant d'entrée. Par défaut, le clavier;
cout	flot courant de sortie. Par défaut, l'écran;
cerr	flot courant de production de message d'erreurs. Par défaut, le même que `cout`;
clog	flot courant de production de message d'erreurs lui aussi, mais les écritures sont temporisées

Pour **lire** des données d'un flot, on utilise l'opérateur **>>**, comme dans:

```
cin >> un_entier >> une_chaine;
```

ou la fonction **get** comme dans:

```
cin.get (un_caractere);
```

Pour **écrire** des données sur un flot, on utilise l'opérateur **<<**, comme dans;

```
cout << "Veuillez fournir un entier et une chaine: ";
```

ou la fonction **put** comme dans:

```
cout.put (un_caractere);
```

Les opérateurs d'*insertion dans un flot* **<<** et d'*extraction d'un flot* **>>** travaillent **à plus haut niveau** que `get` et `put`, lesquels travaillent au niveau des caractères. Nous verrons la différence entre >> et `get` dans les exemples ci-dessous.

Notons que ces opérateurs retournent une référence sur le flot concerné, ce qui permet de **cascader** les opérations successives sur un flot donné.

Lorsque les écritures sur un flot sont **temporisées** (buffered en anglais), les caractères n'y sont effectivement écrits que lorsqu'un tampon (buffer en anglais) est plein. C'est le cas de `clog` mais aussi de `cout`.

Si l'on veut garantir qu'une écriture ait effectivement été faite avant d'exécuter une lecture, pour l'emploi interactif d'un programme par exemple, on peut forcer l'écriture de tout le contenu du tampon à l'aide de la fonction `flush` comme dans:

```
flush (cout);
```

ou de manière équivalente par:

```
cout.flush ();
```

ou encore par:

```
cout << flush;
```

ce qui a pour effet de vider le tampon d'écriture associé à `cout`.

Voici un premier exemple illustrant ces possibilités:

```
#include <stream.h>

main()
     {
     char      caractere_courant;


     cout << "C'est parti…\n";

     while (cin.get (caractere_courant))
          cout.put (caractere_courant);
          // cout << caractere_courant;ferait aussi l'affaire

     cout << "C'est fini.\n";
     }
```

Si l'on applique ce programme avec un flot `cin` contenant:

```
Voici la première…
la seconde…
et la dernière ligne de données
```

on obtient comme résultat sur `cout`:

```
C'est parti…
Voici la première…
la seconde…
et la dernière ligne de données
C'est fini.
```

Notons que ces deux flots peuvent être connectés au clavier et à l'écran, mais aussi à des fichiers selon l'environnement dans lequel ce programme est exécuté.

L'opérateur d'extraction d'un flot **>>** a la particularité d'**ignorer les espaces et les fins de lignes, mais pas les caractères de contrôle**. Ainsi, si l'on applique au flot traité par le premier exemple ci-dessus le second exemple:

```
#include <stream.h>

main()
     {
     char      caractere_courant;

     cout << "C'est parti…\n";
     while (cin >> caractere_courant)
          cout << caractere_courant;

     cout << "C'est fini.\n";
     }
```

on obtient le résultat:

```
C'est parti…
Voicilapremière…laseconde…etladernièrelignededonnéesC'est fini.
```

où l'on voit que les espaces et fins de lignes ont été perdus lors de la lecture du flot.

On peut aussi lire des chaînes de caractères au moyen de >> comme dans le troisième exemple:

```
#include <stream.h>

main()
     {
     char      chaine_courante [256];      // pourrait déborder !!!

     cout << "C'est parti…\n";
     while (cin >> chaine_courante)
          cout << chaine_courante << "\n";

     cout << "C'est fini.\n";
     }
```

Appliqué au même flot que précédemment, ce programme fournit comme résultat:

```
C'est parti…
Voici
la
première…
la
seconde…
et
la
dernière
ligne
de
données
C'est fini.
```

On voit là encore que les espaces et les fins de lignes sont ignorés par >>, ce qui permet de lire tout à tour tous les 'mots' contenus dans le flot.

2.28 Fichiers séquentiels de caractères

La librairie `<fstream.h>` fournit une gestion de fichiers séquentiels de caractères assez puissante, basée sur la gestion des flots. On y trouve essentiellement les types `ifstream`, `ofstream` et `fstream`, décrivant respectivement les fichiers en lecture, en écriture et en lecture/écriture.

Voici un exemple illustrant la gestion de fichiers:

```
#include <fstream.h>
#include <stream.h>

main()
     {
     //   On crée le fichier "fichier_1"

     const     char * fichier_1 = "Fichier1.txt";


     ofstream fichier_ecriture (fichier_1);
     if (! fichier_ecriture)
          cout <<
               form (
                    "Pas pu ouvrir %s pour l'écriture\n",
                    fichier_1
                    );


     fichier_ecriture << "Voici la première…\n";
     fichier_ecriture << 319 << ", la seconde…\n";
     fichier_ecriture << "et la dernière ligne de données\n";
     fichier_ecriture.close ();


     //   On relit le fichier "fichier_1" pour créer "fichier_2"


     const     char * fichier_2 = "Fichier2.txt";


     ifstream fichier_lecture (fichier_1);
     if (! fichier_lecture)
          cout <<
               form (
                    "Pas pu ouvrir %s pour la lecture\n",
                    fichier_1
                    );


     ofstream nouveau_fichier (fichier_2);
     if (! nouveau_fichier)
          cout <<
               form (
                    "Pas pu ouvrir %s pour l'écriture\n",
                    fichier_2
                    );
```

```
char      caractere_courant;


nouveau_fichier << "C'est parti…\n";

while (fichier_lecture.get (caractere_courant))
     nouveau_fichier.put (caractere_courant);

nouveau_fichier << "C'est fini.\n";
}
```

Ce programme crée un fichier `Fichier1.txt` contenant:

```
Voici la première…
319, la seconde…
et la dernière ligne de données
```

puis le relit caractère par caractère pour créer le fichier `Fichier2.txt` contenant quant à lui:

```
C'est parti…
Voici la première…
319, la seconde…
et la dernière ligne de données
C'est fini.
```

On note dans cet exemple l'emploi de l'opérateur `!`, qui fait l'objet d'une *surcharge sémantique* pour accepter un flot lié à un fichier en paramètre. Il indique alors si la **dernière opération sur le fichier** a réussi.

Il existe aussi une fonction `open` duale de `close`, pouvant s'employer comme dans:

```
un_flot_fichier.open (une_chaine);
```

et qui ouvre un fichier donné par son nom dans `une_chaine` pour le lier au flot `un_flot_fichier`.

3 Programmation orientée objets

Ce chapitre présente le corps principal de l'extension qui fait passer de C à C++. Les fonctionnalités correspondantes n'ont pas d'équivalent direct en C.

3.1 Notion d'objet

Fondamentalement, un objet peut être vu de deux points de vue complémentaires:

- un objet est un **module à instances multiples**, le mot module étant pris au sens de Modula-2;
- un objet est un **enregistrement** au sens des `record` de Pascal ou des `struct` de C, **contenant les procédures et fonctions permettant de manipuler les données stockées dans l'objet**.

A y regarder de plus près, ces deux points de vue n'en font qu'un, puisqu'un module Modula–2 est précisément le regroupement de données locales au module et de procédures et fonctions permettant de les manipuler.

Le mot clé réservé C++ `class`, qui vient de Simula 67, introduit syntaxiquement une déclaration de module à instances multiples: il sert à définir une **classe d'objets**, comme par exemple la classe des dates ou celle des arbres de recherche contenant des nombres réels. Chacun de ces objets est par définition une **instance** - un *exemplaire* - de cette classe.

Le lecteur aura remarqué qu'il ne s'agit là que d'un vocabulaire différent pour des choses connues par ailleurs:

- **une classe d'objets est un type**;
- **les instances de cette classe sont les valeurs de ce type**.

Ainsi le nombre `3.141592` est une instance de la classe des nombres réels - une valeur du type réel.

On appelle **instanciation** le processus de création des instances d'une classe. Ceci est fait en C++ soit:

- **en définissant statiquement une variable de ce type classe**, globale à un fichier, locale à une fonction, à un bloc dans une fonction ou membre d'une autre classe;
- **en créant dynamiquement une instance à l'aide de l'opérateur `new`**, analogue au `new` de Pascal et au `malloc` de C.

☞ Les instances de classes sont C++ **des variables comme les autres**: elles sont simplement d'un type 'classe' au lieu d'être d'un type prédéfini comme `int` ou `float`.

En plus de l'aspect 'modules à instances multiples', l'orientation objets permet de construire des **hiérarchies de classes - de types** - dans lesquelles une classe hérite des champs et des procédures et fonctions d'autres classes. Cette manière de **factoriser les champs et le code** s'appelle l'**héritage**.

3.2 Modules à instances multiples

C++ permet de déclarer des classes et des sous-classes comme Simula 67, avec toutefois une différence majeure: **en C++, une instance d'une classe n'est pas nécessairement allouée dynamiquement**, ce qui a des répercussions assez importantes sur tout l'aspect orienté objets du langage. En fait, **une classe C++ est très voisine d'une structure** (`struct`), **à la protection d'accès près**, ce qui sera précisé plus loin. Ceci illustre d'ailleurs le fait qu'une classe n'est fondamentalement qu'un enregistrement contenant textuellement les procédures et fonctions permettant de manipuler les instances. Bien entendu, la notion d'héritage qui s'y ajoute pour constituer l'orientation objets est inconnue en C.

Dans une classe C++, on définit les champs, appelés **membres** (members en anglais), et les **fonctions membres** (member functions en anglais), en précisant leur **visibilité**. **Par défaut**, ces identificateurs sont **privés** à la classe (**`private`** en anglais), et ne sont visibles que par les fonctions elle-mêmes membres de la classe, comme `+` et `ecrire` dans l'exemple ci-dessous.

Cet aspect 'privé par défaut' est d'ailleurs **la seule différence entre une classe et une structure**, dans laquelle les identificateurs sont **par défaut visibles** en dehors de la structure.

Il est possible dans les déclarations de classes et de structures de ne pas appliquer la visibilité par défaut et de **préciser la visibilité des membres et fonctions membres** en employant un des mots clés `private`, `protected` ou `public`. La sémantique en est la suivante:

- **`private`**:

  indique que les champs dont les définitions suivent ne peuvent être utilisés *que par les fonctions membres et les classes et opérations amies* (mot clé `friend`) de cette *seule* classe, à l'exception des sous-classes;
- **`protected`**:

  indique que les définitions qui suivent sont également visibles dans les classes descendant par héritage de celle en cours de déclaration: les fonctions membres des sous-classes les voient *comme leurs propres déclarations privées*, tandis que toutes les autres fonctions ne les voient pas, au même titre que les déclarations privées;
- **`public`**:

indique qu'il n'y a *pas de restriction de visibilité*, et que l'on peut utiliser les champs de la classe ou structure depuis le corps de toute fonction dans laquelle le nom de la classe est lui-même visible.

Notons encore que l'héritage, que l'on verra en pratique plus loin, se fait en C++ au moyen des mêmes mots clés `private`, `protected` et `public`, indiquant ainsi quelle visibilité ont les champs des super-classes dans les sous-classes.

3.3 Exemple d'instances multiples: interface

Considérons commes exemple de modules à instances multiples le cas de gestion de dates, qui avait été illustré sans objets au paragraphe 2.8, page 12. Dans cette nouvelle version, le fichier **`Dates.h`** contient:

```
#ifndef __Dates__
#define __Dates__


#include <stream.h>
#include <types.h>


//   Les jours de la semaine
//   -----------------------

typedef enum nom_de_jour
     {dimanche, lundi, mardi, mercredi, jeudi, vendredi, samedi};


ostream            & operator << (ostream & os, nom_de_jour le_jour);


//   Les mois de l'annee
//   -------------------

typedef enum mois
     {
     janvier = 1, fevrier, mars, avril, mai, juin,
     juillet, aout, septembre, octobre, novembre, decembre
     };
ostream            & operator << (ostream & os, mois le_mois);


//   Les dates
//   ---------

typedef long facteur;

//                 Référence:
//                      Financial Calculations for Business
//                      Christian de Lisle
//                      Kogan Page, 1990
//                      ISBN 1-85091-978-X
```

```
class DATE
     // Description d'une date
     {
public:
                    DATE (short le_jour, mois le_mois, short l_annee);
                         //   CONSTRUCTEUR !


     Boolean        date_valide () const;


     char           * sous_forme_de_chaine () const;


     Boolean        date_inf (const DATE & autre_date) const;
     Boolean        date_inf_egale (const DATE & autre_date) const;


     long           diff_date (const DATE & autre_date) const;


     nom_de_jour    jour_de_la_semaine () const;


     Boolean        annee_bissextile () const;
     short          nb_29_fevrier_entre (const DATE & autre_date) const;


     DATE           operator + (int nb_jours) const;
     DATE           operator - (int nb_jours) const;


private:

     short          fJour;
     mois           fMois;
     short          fAnnee;

     facteur        facteur_de_date () const;

     };   //   DATE


inline DATE :: DATE (short le_jour, mois le_mois, short l_annee)
     // CONSTRUCTEUR !
     { fJour = le_jour; fMois = le_mois;   fAnnee = l_annee; }


//   Les annees bissextiles
//   ----------------------

Boolean             annee_bissextile (short annee);

DATE                jour_de_facteur (facteur le_facteur);

#endif __Dates__
```

Dans cet exemple, nous avons tout d'abord défini diverses notions pour la gestion des jours et des mois. A noter la spécification:

```
janvier = 1
```

dans le définition du type `mois`, pour se conformer à l'usage courant.

Ensuite de cela nous avons défini `DATE` comme une **classe** contenant trois membres privés `fJour`, `fMois` et `fAnnee`, ainsi que diverses fonctions et fonctions membres. On remarquera que toutes les fonctions prenant un paramètre de type `DATE` dans le paragraphe 2.8, page 12 et qui sont devenues des fonctions membres comptent désormais **un paramètre de moins**, comme:

```
Boolean DATE :: date_inf (const DATE & autre_date)
```

Ceci s'explique par le fait qu'une instance de `DATE` est désormais un **objet actif**, capable de **répondre au stimulus** '**est-tu inférieure à** `une_autre_date`', ce qui s'écrit:

```
une_date.date_inf (une_autre_date)
```

Syntaxiquement, un des deux arguments d'appel à `date_inf` est simplement **passé en dehors de l'appel**, pour devenir le **destinataire du message** `date_inf (…)`.

On notera les différentes *surcharges sémantiques* de l'opérateur d'insertion dans un flot `<<`, réalisées pour pouvoir écrire aisément différents types de valeurs sur un flot. Ces surcharges retournent toujours une *référence sur le flot dans lequel on vient de faire l'insertion*, ce qui permet de *cascader* les écritures, comme cela est illustré dans le fichier `Main.cp` ci-dessous.

Il est possible de spécifier qu'une fonction membre est une **fonction membre pure**, au sens mathématique du terme, et qu'elle n'a *pas le droit de modifier l'objet qui lance son exécution en réponse à un message*. Cela se fait en plaçant le mot clé réservé `const` après l'entête de la fonction membre dans sa déclaration **et** dans sa définition, comme pour:

```
Boolean DATE :: date_valide () const
```

Il y a un **effet boule de neige** pour ces fonctions membres pures: elle ne peuvent pas elles-mêmes passer l'objet concerné en argument à d'autres fonctions qui pourraient le modifier par référence où via un pointeur. Cela conduit souvent en pratique à devoir déclarer `const` d'autres fonctions, et ainsi de suite!

On ne peut *pas* utiliser cette spécification de fonction pure pour les fonctions ordinaires, non membres d'une classe.

Une fonction membre portant le même nom que la classe à laquelle elle appartient, comme `DATE :: DATE`, est par définition appelée un **constructeur** de cette classe. Disons simplement pour l'instant que ce constructeur est une *procédure* qui sert à donner aux instances de la classe `DATE` un état interne initial connu du programmeur. Le détail du mécanisme mis en œuvre sera présenté au paragraphe 3.7, page 14.

Dans notre cas particulier, le constructeur est déclaré `inline`, pour éviter si possible le prix d'un appel de fonction lors de la mise en place de cet état initial. Ceci implique que

la définition de ce constructeur doit figurer **dans le fichier `Dates.h`**, afin que tous les clients important ce dernier aient connaissance du détail du code à développer en lieu et place d'un appel de fonction. Cette **entorse à la règle de masquage des détails d'implantation** d'une classe est le prix à payer pour obtenir l'efficacité dans ce cas.

On remarquera aussi la surcharge sémantique des opérateurs '+' et '-' comme dans:

```
DATE DATE :: operator + (int nb_jours) const;
```

ce qui permet d'additionner ou soustraire un nombre de jours à une `DATE` pour obtenir une autre `DATE`.

Enfin, la classe `DATE` comporte une **fonction membre privée**:

```
facteur DATE :: facteur_de_date () const;
```

qui n'est donc accessible qu'aux seules fonctions membres de cette classe. Cette **encapsulation** est justifiée dans ce cas par le fait que le détail de l'arithmétique sur les dates et du décompte des 29 février n'a pas à être accessible aux clients de la classe.

Nous renvoyons le lecteur intéressé aux détails des calculs mis en œuvre au libre cité en commentaire dans le fichier `Dates.h` ci-dessus.

3.4 Exemple d'instances multiples: implantation

La définition des différentes fonctions et fonctions membres déclarées au paragraphe précédent est faite dans le fichier **`Dates.cp`** contenant:

```
#include "Dates.h"


#include <math.h>


//   Les jours de la semaine
//   -----------------------

static const char * noms_des_jours [] =    //   un tableau privé
     {
     "dimanche", "lundi", "mardi", "mercredi",
     "jeudi", "vendredi", "samedi"
     };


ostream & operator << (ostream & os, nom_de_jour le_jour)
     {
     os << noms_des_jours [le_jour];
     return os;
     }    // operator <<


//   Les mois de l'annee
//   -------------------

static const char * noms_des_mois [] =     //   un tableau privé
     {
```

```
     "janvier", "fevrier", "mars", "avril", "mai", "juin",
     "juillet", "aout", "septembre", "octobre", "novembre", "decembre"
     };


ostream & operator << (ostream & os, mois le_mois)
     {
     os << noms_des_mois [le_mois];
     return os;
     }    // operator <<


static const int jours_par_mois [12] =     //   un tableau privé
     /* jan fev mar avr mai jun jul aou sep oct nov dec */
     {  31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31  };


//   Les dates
//   ---------

Boolean DATE :: date_valide () const
     {
     return
          (fMois >= 1)
               &&
          (fMois <= 12)
               &&
          (fJour <= jours_par_mois [fMois - 1])
               &&
          ((fJour == 29) ? annee_bissextile () : true);
     }    //   DATE :: date_valide


char * DATE :: sous_forme_de_chaine () const
     {
     return
          form ("(Jour: %d, Mois: %d, Annee: %d)", fJour, fMois, fAnnee);
     }


facteur DATE :: facteur_de_date () const
     {
     short     Yp =
                    fMois > fevrier ? fAnnee : fAnnee - 1;
     short     Mp =
                    fMois > fevrier ? fMois + 1 : fMois + 13;

     return
          long (365.25 * Yp) + long (30.6 * Mp) + fJour;
     }    //   DATE :: facteur_de_date


Boolean DATE :: date_inf (const DATE & autre_date) const
     { return facteur_de_date () < autre_date.facteur_de_date (); }
```

```
Boolean DATE :: date_inf_egale (const DATE & autre_date) const
     { return facteur_de_date () <= autre_date.facteur_de_date (); }


long DATE :: diff_date (const DATE & autre_date) const
     { return facteur_de_date () - autre_date.facteur_de_date (); }


inline float frac (float r)
     { return r - floor (r); }


inline long round (float r)
     { return frac (r) < 0.5 ? floor (r) : ceil (r); }


nom_de_jour DATE :: jour_de_la_semaine () const
     {
     short     l_annee_inter =
                    fMois > fevrier ? fAnnee : fAnnee - 1;

     return
          nom_de_jour (
               round (
                    7 * frac (
                         (
                         facteur_de_date () -
                         short (
                              0.75 * (short (l_annee_inter / 100.0) - 7) )
                              )
                         / 7.0
                    )
               ));
     }    //   DATE :: jour_de_la_semaine


short DATE :: nb_29_fevrier_entre (const DATE & autre_date) const
     {
     short     l_annee_1 =
                    fMois > fevrier ? fAnnee : fAnnee - 1;
     short     l_annee_2 =
                    autre_date.fMois > fevrier
                         ? autre_date.fAnnee
                         : autre_date.fAnnee - 1;

     return fabs (short (l_annee_1 / 4.0) - short (l_annee_2 / 4.0));
     }


Boolean DATE :: annee_bissextile () const
     { return :: annee_bissextile (fAnnee); }


DATE DATE :: operator + (int nb_jours) const
     { return jour_de_facteur (facteur_de_date () + nb_jours); }
```

```
DATE DATE :: operator - (int nb_jours) const
     { return jour_de_facteur (facteur_de_date () - nb_jours); }


//   Les annees bissextiles
//   ----------------------


Boolean annee_bissextile (short annee)
     //   Seuls les siècles multiples de 400 sont bissextiles!
     {
     return
          (annee % 4 == 0)
               &&
          ( (annee % 100 == 0) ? (annee % 400 == 0) : true );
     }


DATE jour_de_facteur (facteur le_facteur)
     {
     short     l_annee_inter =
                    short ((le_facteur - 122.4) / 365.25);
     mois      le_mois_inter =
                    mois ((le_facteur - long (365.25 * l_annee_inter))
                    / 30.6);


     short     le_jour =
                    le_facteur -
                    long (365.25 * l_annee_inter)
                    - short (30.6 * le_mois_inter);
     mois      le_mois =
          mois (le_mois_inter < 14
               ? le_mois_inter - 1
               : le_mois_inter - 13);


     return DATE    // utilise le CONSTRUCTEUR
          (
          le_jour,
          le_mois,
          le_mois > fevrier ? l_annee_inter : l_annee_inter + 1
          );
     }    //   jour_de_facteur
```

On peut remarquer dans ce fichier l'emploi de **variables privées au module** `Dates`, comme:

```
static char    * noms_des_jours []
```

Rappelons que le mot clé pré-défini `static` signifie que `noms_des_jours` est un **identificateur global à visibilité restreinte** au seul module `Dates`. Sans le mot clé `static`, ce tableau pourrait être utilisé par un autre module compilé indépendamment de `Dates` et lié avec lui par l'éditeur de liens (linker en anglais) pour constituer un programme exécutable.

On retrouve dans le corps de la fonction membre `DATE :: date_inf` le fait qu'un des paramètres est maintenant devenu le *destinataire* du message. C'est donc lui qui est concerné par l'appel à `facteur_de_date` figurant à gauche de l'opérateur <:

```
Boolean DATE :: date_inf (const DATE & autre_date) const
     {
     return
          facteur_de_date () <
               autre_date.facteur_de_date ();
     }
```

Certains aiment mettre en évidence les envois de message depuis le corps d'une méthode à l'objet même qui est en train d'exécuter cette méthode en réponse à un message. Cela peut être fait dans l'exemple ci-dessus à l'aide du mot clé **this**:

```
return
     this -> facteur_de_date () <
          autre_date.facteur_de_date ();
```

Enfin, la surcharge sémantique de l'opérateur `'+'` utilise la fonction membre privée `facteur_de_date` pour calculer son résultat:

```
DATE DATE :: operator + (int nb_jours) const
     { return jour_de_facteur (facteur_de_date () + nb_jours); }
```

Une fonction C++ peut retourner un résultat de n'importe quel type, donc une `DATE`.

3.5 Exemple d'instances multiples: utilisation

On peut utiliser le module `Dates` depuis un programme principal `comme Main.cp` contenant:

```
#include "Dates.h"
#include <stream.h>

main ()
     {
     DATE      date_debut    (25, decembre, 1976);
                                  //   utilise le constructeur
     DATE      date_fin      ( 1,  janvier, 2000);
                                  //   utilise le constructeur


     cout <<
          "date_fin - date_debut = " <<
          date_fin.diff_date (date_debut) <<
          "\n\n";


     cout <<
          "date_debut.nb_29_fevrier_entre (date_fin) = " <<
          date_debut.nb_29_fevrier_entre (date_fin) <<
```

```
        "\n\n";


    cout <<
        "annee_bissextile (1984) = " <<
        (annee_bissextile (1984) ? "vrai" : "faux") <<
        "\n";

    cout <<
        "annee_bissextile (1985) = " <<
        (annee_bissextile (1985) ? "vrai" : "faux") <<
        "\n";

    cout <<
        "annee_bissextile (1900) = " <<
        (annee_bissextile (1900) ? "vrai" : "faux") <<
        "\n";

    cout <<
        "annee_bissextile (2000) = " <<
        (annee_bissextile (2000) ? "vrai" : "faux") <<
        "\n";
    cout << "\n";


    cout <<
        "DATE (28, fevrier, 1984).date_valide () = " <<
        (DATE (28, fevrier, 1984).date_valide () ? "vrai" : "faux") <<
        "\n";

    cout <<
        "DATE (29, fevrier, 1984).date_valide () = " <<
        (DATE (29, fevrier, 1984).date_valide () ? "vrai" : "faux") <<
        "\n";

    cout <<
        "DATE (30, fevrier, 1984).date_valide () = " <<
        (DATE (30, fevrier, 1984).date_valide () ? "vrai" : "faux") <<
        "\n";

    cout <<
        "DATE (28, fevrier, 1985).date_valide () = " <<
        (DATE (28, fevrier, 1985).date_valide () ? "vrai" : "faux") <<
        "\n";

    cout <<
        "DATE (29, fevrier, 1985).date_valide () = " <<
        (DATE (29, fevrier, 1985).date_valide () ? "vrai" : "faux") <<
        "\n";

    cout <<
        "DATE (30, fevrier, 1985).date_valide () = " <<
        (DATE (30, fevrier, 1985).date_valide () ? "vrai" : "faux") <<
        "\n";
    cout << "\n";
```

```
    cout <<
         "date_debut.jour_de_la_semaine () = " <<
         date_debut.jour_de_la_semaine () << "\n";

    cout <<
         "date_fin.jour_de_la_semaine () = " <<
         date_fin.jour_de_la_semaine () << "\n";

    cout <<
         "DATE (1, janvier, 1806).jour_de_la_semaine () = " <<
         DATE (1, janvier, 1806).jour_de_la_semaine () << "\n\n";


    cout <<
         "jour_de_facteur (730563) = " <<
         jour_de_facteur (730563).sous_forme_de_chaine () << "\n";

    cout <<
         "date_fin + 1 = " <<
         (date_fin + 1).sous_forme_de_chaine () << "\n";

    cout <<
         "date_fin - 1 = " <<
         (date_fin - 1).sous_forme_de_chaine () << "\n";
    }
```

Ce programme produit comme résultats:

```
date_fin - date_debut = 8407

date_debut.nb_29_fevrier_entre (date_fin) = 5


annee_bissextile (1984) = vrai
annee_bissextile (1985) = faux
annee_bissextile (1900) = faux
annee_bissextile (2000) = vrai


DATE (28, fevrier, 1984).date_valide () = vrai
DATE (29, fevrier, 1984).date_valide () = vrai
DATE (30, fevrier, 1984).date_valide () = faux
DATE (28, fevrier, 1985).date_valide () = vrai
DATE (29, fevrier, 1985).date_valide () = faux
DATE (30, fevrier, 1985).date_valide () = faux


date_debut.jour_de_la_semaine () = samedi
date_fin.jour_de_la_semaine () = samedi
DATE (1, janvier, 1806).jour_de_la_semaine () = mercredi


jour_de_facteur (730563) = (Jour: 1, Mois: 1, Annee: 2000)
date_fin + 1 = (Jour: 2, Mois: 1, Annee: 2000)
date_fin - 1 = (Jour: 31, Mois: 12, Annee: 1999)
```

Sans entrer dans tous les détails à ce stade, notons que la variable `date_debut`, locale à la fonction `main`, contiendra initialement la date du `25 décembre 1976`, comme indiqué par l'emploi du **constructeur** `DATE :: DATE` dans:

```
DATE     date_debut         (25, decembre, 1976);
```

On remarque aussi l'emploi d'une **instance flottante** du type `DATE` dans l'extrait ci-dessous: elle n'est **pas stockée dans une variable**, mais elle est une expression, de la même manière que `32+5` pour le type entier:

```
DATE     (28, fevrier, 1984).date_valide ()
```

Le constructeur est là aussi utilisé pour donner un état initial à cette instance, et on peut lui appliquer toutes les fonctions membres connues pour une `DATE`.

Une autre instance flottante est retournée par la fonction `jour_de_facteur` puis mise sous forme de chaîne de caractères dans:

```
jour_de_facteur (730563).sous_forme_de_chaine ()
```

comme d'ailleurs la valeur - l'instance - du type `DATE` retournée par '+' dans l'extrait ci-dessous.

```
(date_fin + 1).sous_forme_de_chaine ()
```

3.6 Accès aux membres privés d'une classe

Dans l'exemple des paragraphes précédents, nous avons défini une fonction membre:

```
char * DATE :: sous_forme_de_chaine () const
     {
     return
          form (
               "(Jour: %d, Mois: %d, Annee: %d)",
               fJour, fMois, fAnnee );
     }    //   DATE :: sous_forme_de_chaine
```

pour pouvoir imprimer une date, ce qui peut alors se faire au moyen de:

```
cout << une_date.sous_forme_de_chaine ();
```

En fait, il est possible de surcharger sémantiquement l'opérateur d'insertion dans un flot `<<` pour qu'il puisse écrire des dates sans passer par la fonction membre ci-dessus. On se heurte toutefois alors au fait que les membres `fJour`, `fMois` et `fAnnee` sont *privés*, tandis que l'opérateur << est une fonction globale, et donc *pas une fonction membre de* `DATE`. Pour pouvoir accéder à ces membres privés, on peut s'y prendre de deux manières:

- soit on ajoute à `DATE` des fonctions membres permettant de **consulter la valeur des membres privés**, comme:

  ```
  Boolean DATE :: annee () const
       { return fAnnee; }
  ```

et on place dans le fichier `Dates.cp` la définition:

```
ostream & operator << (ostream & os, const DATE & une_date)
     {
     os <<
          form (
               "(Jour: %d, Mois: %d, Annee: %d)",
               jour (), mois (), annee () );
     return os;
     }    // operator <<
```

- soit on place dans la déclaration de `DATE`, dans le fichier `Dates.h`, la déclaration de **fonction amie** (friend function en anglais):

```
friend
     ostream & operator <<
          (ostream & os, const DATE & une_date);
```

et dans le fichier `Dates.cp` sa définition:

```
ostream & operator << (ostream & os, const DATE & une_date)
     {
     os <<
          form (
               "(Jour: %d, Mois: %d, Annee: %d)",
               fJour, fMois, fAnnee );
     return os;
     }    // operator <<
```

☞ **Dans les deux manière de faire ci-dessus, on dévoile partiellement aux clients de la classe `DATE` les détails de son implantation**.

Nous verrons dans la suite de ce chapitre les deux techniques mises en œuvre.

La déclaration '`friend ostream & operator <<`' indique que l'opérateur << n'est pas une fonction membre de la classe `DATE`, mais qu'il peut utiliser les *membres privés des instances de cette classe* librement.

☞ L'emploi de `friend` est une **relaxation de la règle de portée textuelle des langages à structure de blocs**, car il donne aux fonctions amies les mêmes privilèges que ceux des fonctions membres. Il est de ce fait assez controversé.

En pratique, il ne semble à conseiller que pour réaliser des surcharges sémantiques des opérateurs comme les opérations arithmétiques et les d'entrées-sorties.

3.7 Constructeurs et destructeurs

Une caractéristique de C++ est que l'on peut associer à chaque classe des **constructeurs** et au plus un**destructeur**:

- un **constructeur `une_classe :: une_classe` est appelé implicitement pour traiter chaque instance du type `une_classe` qui vient de naître et qui n'est pas initialisée au moyen de:**

```
une_classe une_instance = une_expression;
```

C'est en quelque sorte le **cri primal de l'instance**;

- le **destructeur** `~ une_classe :: une_classe`, s'il existe, est appelé implicitement pour **traiter chaque instance du type** `une_classe` **qui va mourir**: c'est en quelque sorte les **dernières volontés de l'instance**;

La terminologie C++ pourrait faire croire que c'est le constructeur qui met l'instance au monde et le destructeur qui met fin à ses jours. Il n'en est rien: **l'instance est déjà née lorsqu'un constructeur lui est appliqué**, et **elle ne meurt qu'après l'exécution du destructeur**.

En cas d'absence de constructeurs définis par le programme, le compilateur peut créer un constructeur par défaut, selon les besoins.

On ne peut pas indiquer de type retourné pour les constructeurs ni pour les destructeurs.

Un **constructeur par défaut** est un constructeur *sans paramètres*, ou *pouvant être appelé sans paramètres* en vertu des valeurs par défaut des paramètres, comme on le verra dans l'exemple ci-dessous.

Rappelons ce que nous avons vu au paragraphe 2.10, page 19, à savoir que:

- une variable globale est implantée **statiquement**: elle naît juste avant le début de l'exécution de la fonction `main`, et meurt juste après la fin de cette exécution;
- une variable paramètre d'une fonction ou déclarée localement à une fonction ou à un bloc interne à une fonction naît **automatiquement** à l'entrée dans le bloc où elle est déclarée, et meurt automatiquement à la sortie de ce bloc;
- une variable créée **dynamiquement** par l'opérateur `new` naît à ce moment, et fait l'objet d'un constructeur juste après. Si on la tue au moyen de l'opérateur `delete`, elle est sujette au destructeur éventuel juste avant sa mort.

Nous verrons dans les paragraphes suivants des exemples de l'emploi de `new` et `delete`, et les détails du fonctionnement de ces deux opérateurs seront présentés au Aspects avancés du langage.

Voici un exemple illustrant ces différentes **durées de vie d'instances**, basé sur les nombres complexes chers aux mathématiciens et électriciens. Le fichier `complexes.h` contient:

```
//   Gestion de complexes en C++

#ifndef __Complexes__
#define __Complexes__

#include <stream.h>

class complexe
     {
public:
```

```
                        complexe ( float r = 0, float i = 0);
                             //   un constructeur pouvant être
                             //   utilisé comme constructeur par défaut

                        complexe ( float r, float i, short bidon );
                             //   un constructeur pour "le_nombre_i"

                        ~ complexe ();
                             //   LE destructeur

     friend ostream     & operator << (
                             ostream   & o,
                             complexe  le_complexe );

     void               ecrire ();

     friend complexe    operator + (            // déclaration
                             complexe  le_complexe_1,
                             complexe  le_complexe_2 );

     friend complexe    operator + (
                             float     le_reel,
                             complexe  le_complexe );

     friend complexe    operator + (
                             complexe  le_complexe,
                             float     le_reel );

     friend complexe    operator * (
                             complexe  le_complexe_1,
                             complexe  le_complexe_2 );

private :

     float                   fRe, fIm;

     }; // complexe

extern const complexe le_nombre_i;    // déclaration

#endif __Complexes__
```

Le fichier `complexes.cp` contient la définition des fonctions membres de la classe `complexe`, soit:

```
#include "complexes.h"

complexe :: complexe ( float r, float i)
     //   un constructeur pouvant être
     //   utilisé comme constructeur par défaut
     {
     fRe = r;
     fIm = i;

     cout <<
          "\t\t+++\ton construit (" <<
          fRe << " | " << fIm << ")" <<
          form (", adresse %x", this) << "\n";
     }    //   complexe :: complexe


complexe :: complexe ( float r, float i, short bidon )
     //   un constructeur pour "le_nombre_i"
     //   il ne fait pas d'impression
     { fRe = r; fIm = i; }


complexe :: ~ complexe ()
     // LE destructeur
     {
     cout <<
          "\t\t---\ton détruit (" <<
          fRe << " | " << fIm << ")" <<
          form (", adresse %x", this) << "\n";
     }


ostream & operator << (
     ostream   & o,
     complexe  le_complexe
     )
     {
     if (le_complexe.fRe != 0.0)
          {
          o << le_complexe.fRe;
          if (le_complexe.fIm != 0.0)
               {
               if (le_complexe.fIm > 0.0)
                    o << " +";
               o << le_complexe.fIm << " i";
               }
          }

     else
          {
          if (le_complexe.fIm != 0.0)
               {
               if (le_complexe.fIm > 0.0)
```

```
                    o << " +";
               o << le_complexe.fIm << " i";
               }
          else
               o << "0";
          }
     return( o );
     }    //   operator <<


void complexe :: ecrire ()
     { cout << * this; }


complexe operator + (             // définition
     complexe  le_complexe_1,
     complexe  le_complexe_2
     )
     {
     complexe  res (
                    le_complexe_1.fRe + le_complexe_2.fRe,
                    le_complexe_1.fIm + le_complexe_2.fIm );
     return res;
     }


complexe operator + (
     float     le_reel,
     complexe  le_complexe
     )
     {
     return complexe (
          le_complexe.fRe + le_reel,
          le_complexe.fIm );
     }    //   operator +


complexe operator + (
     complexe  le_complexe,
     float     le_reel
     )
     {
     return complexe (
          le_complexe.fRe + le_reel,
          le_complexe.fIm );
     }    //   operator +


complexe operator * (
     complexe  le_complexe_1,
     complexe  le_complexe_2
     )
     {
     return complexe (
          le_complexe_1.fRe * le_complexe_2.fRe -
               le_complexe_1.fIm * le_complexe_2.fIm,
```

```
            le_complexe_1.fRe * le_complexe_2.fIm +
                le_complexe_1.fIm * le_complexe_2.fRe );
        }    //   operator *

static const complexe le_nombre_i ( 0, 1, -3 );     // définition
```

Le premier constructeur déclaré peut être employé comme **constructeur par défaut**, puisqu'il est possible de l'appeler sans arguments. On a déclaré un *constructeur spécial* pour `le_nombre_i`: cette variable étant globale et donc allouée statiquement, elle est construite avant de début de l'exécution de la fonction `main`, à un moment où le flot de sortie `cout` n'est pas encore disponible: on n'aura donc pas de trace de cette construction. La destruction de `le_nombre_i`, quant à elle, intervient après la fin de l'exécution de `main`, et la destruction ne produit pas de trace parce que `cout` est déjà déconnecté du programme à ce moment.

On peut utiliser la classe `complexe` avec un programme principal `Main.cp` comme:

```
#include "complexes.h"

main ()
    {
    cout << "DEBUT de 'MAIN'\n\n";


    complexe complexe_1 = le_nombre_i;
        // initialisation


    complexe complexe_2 = le_nombre_i * le_nombre_i;
        // initialisation


    complexe complexe_3;
        // a besoin du constructeur par défaut!


    const complexe le_nombre_i_bis ( 0, 1 );
        // a besoin du constructeur à deux paramètres float


    // impression des adresses

    cout << "\n";
    cout << form(
        "adresse(%s) = %x", "le_nombre_i", & le_nombre_i ) << "\n";
    cout << form("adresse(%s) = %x", "complexe_1", & complexe_1) <<
"\n";
    cout << form("adresse(%s) = %x", "complexe_2", & complexe_2) <<
"\n";
    cout << form("adresse(%s) = %x", "complexe_3", & complexe_3) <<
"\n\n";


    // c'est parti!
```

```
cout << "complexe_3 initial = " << complexe_3 << "\n\n";


complexe_3 = 1.1 + 2.2 + complexe_2 + 3.0;
complexe_3 = complexe_3 + complexe (-3.0, -4.5);


     {         // BLOC INTERNE

     cout << "\nENTREE DE BLOC\n";
     complexe complexe_4 (12.5, 77);
     cout << "complexe_4 = " << complexe_4 << "\n";
     cout << "FIN DE BLOC\n\n";

     }


cout << "\n";
cout << "complexe_1 = " << complexe_1 << "\n";
cout << "complexe_2 = " << complexe_2 << "\n";
cout << "complexe_3 = " << complexe_3 << "\n\n";


cout << "complexe_3 * complexe_3 = ";
(complexe_3 * complexe_3).ecrire ();
cout << "\n\n";


complexe * pointeur_sur_complexe = new complexe (3.2, 5);
cout <<
     "\n* pointeur_sur_complexe = " <<
     * pointeur_sur_complexe << "\n\n";
delete pointeur_sur_complexe;

cout << "\nFIN de 'MAIN'\n\n";
}
```

☞ Il y a manifestement **surcharge sémantique** pour les constructeurs d'une classe donnée. Là encore, toute ambigüité constitue une erreur de sémantique statique, dépistée à la compilation.

Pour pouvoir imprimer des valeurs du type `complexe`, il nous suffit d'effectuer une surcharge sémantique de l'opérateur <<. On voit l'intérêt de la surcharge sémantique dans ce cas.

☞ Le lecteur pourra se demander pourquoi nous n'avons pas défini des *fonctions membres comme '+' et '*' dans la classe* `complexe`: la raison est que seules les versions de ces opérations dont le premier argument est lui-même du type `complexe` s'y prêteraient valablement.

On ne peut en effet pas envoyer un message comme '+' à un entier ou à un flottant, puisque les types correspondants ne sont pas des types 'classe'. Par souci

d'homogénéïté, il a donc semblé préférable d'utiliser systématiquement des surcharges sémantiques des opérateurs prédéfinis.

Il est en fait **possible se surcharger sémantiquement la plupart des opérateurs de C++, et même l'affectation et l'appel de procédure**. Nous en verrons des applications au Aspects avancés du langage.

Dans la définition de l'opérateur:

```
complexe operator + (            // définition
     complexe  le_complexe_1,
     complexe  le_complexe_2
     )
     {
     complexe  res (
                    le_complexe_1.fRe + le_complexe_2.fRe,
                    le_complexe_1.fIm + le_complexe_2.fIm );
     return res;
     }    //   operator +
```

il suffit de déclarer la variable locale `res` avec les valeurs voulues pour `res.fRe` et `res.fIm`, ce qui provoque l'appel *implicite* du constructeur à deux paramètres `float`, et de retourner la valeur de `res` comme résultat de la fonction. En fait, la variable res est strictement superflue, comme le montre la définition de l'opérateur:

```
complexe operator + (
     float     le_reel,
     complexe  le_complexe
     )
     {
     return complexe (
          le_complexe.fRe + le_reel,
          le_complexe.fIm );
     }    //   operator +
```

Notons encore l'instruction:

```
(complexe_3 * complexe_3).ecrire ();
```

dans laquelle on utilise l'**instance flottante** `complexe_3 * complexe_3`, du type `complexe`, à laquelle on applique la fonction membre `ecrire`.

Enfin, la création dynamique d'une instance de complexe par:

```
complexe * pointeur_sur_complexe = new complexe (3.2, 5);
```

cause l'*appel implicite du constructeur* à deux paramètres `float`, tandis que sa mort par:

```
delete pointeur_sur_complexe;
```

provoque l'*appel implicite du destructeur* `~ complexe :: complexe`.

On obtient par ce programme les résultats suivants, les commentaires *en italique* ayant été ajoutés après coup pour nos besoins:

```
DEBUT de 'MAIN'

         +++  on construit (-1 | 0), adresse c9e4c8
                                //  res de '*', instance flottante
         ---  on détruit (-1 | 0), adresse c9e4c8
                                //  res de '*', instance flottante
         +++  on construit (0 | 0), adresse c9e550
                                //  constructeur par défaut, complex_3
         +++  on construit (0 | 1), adresse c9e558
                                //  constructeur de "le_nombre_i_bis"


adresse(le_nombre_i) = 9d6f22
adresse(complexe_1) = c9e540
adresse(complexe_2) = c9e548
adresse(complexe_3) = c9e550


complexe_3 initial = 0


         +++  on construit (2.3 | 0), adresse c9e444
                                // res de '+' (1.1+2.2+complexe_2)
         ---  on détruit (2.3 | 0), adresse c9e444
                                // res de '+' (1.1+2.2+complexe_2)
         +++  on construit (5.3 | 0), adresse c9e44e
                                // res de '+' (1.1+2.2+complexe_2+3.0)
         ---  on détruit (5.3 | 0), adresse c9e44e
                                // res de '+' (1.1+2.2+complexe_2+3.0)
         +++  on construit (-3 | -4.5), adresse c9e538
                                // complexe (-3.0, -4.5)
         +++  on construit (2.3 | -4.5), adresse c9e43e
                                // res de '+' (complexe_3+complexe(-3.0, -4.5))
         ---  on détruit (2.3 | -4.5), adresse c9e43e
                                // res de '+' (complexe_3+complexe(-3.0, -4.5))
         ---  on détruit (-3 | -4.5), adresse c9e538
                                // complexe(-3.0, -4.5)


ENTREE DE BLOC
         +++  on construit (12.5 | 77), adresse c9e528
                                // complexe_4
complexe_4 = 12.5 +77 i
FIN DE BLOC


         ---  on détruit (12.5 | 77), adresse c9e528
                                // complexe_4


complexe_1 =  +1 i
complexe_2 = -1
complexe_3 = 2.3-4.5 i
```

```
complexe_3 * complexe_3 = +++on construit (-14.96 | -20.7), adresse c9e3c2
                          // res de '*'
         ---  on détruit (-14.96 | -20.7), adresse c9e3c2
                          // res de '*'
-14.96-20.7 i

         +++  on construit (3.2 | 5), adresse 9d8ff4
                          // * pointeur_sur_complexe

* pointeur_sur_complexe = 3.2 +5 i

         ---  on détruit (3.2 | 5), adresse 9d8ff4
                          // * pointeur_sur_complexe

FIN de 'MAIN'

         ---  on détruit (-14.96 | -20.7), adresse c9e530
                          // complexe_3*complexe_3
         ---  on détruit (0 | 1), adresse c9e558
                          // le_nombre_i_bis
         ---  on détruit (2.3 | -4.5), adresse c9e550
                          // complexe_3
         ---  on détruit (-1 | 0), adresse c9e548
                          // complexe_2
         ---  on détruit (0 | 1), adresse c9e540
                          // complexe_1
```

On remarquera les points suivants:

- **s'il existe un constructeur au moins, toute instance qui naît est soit construite par un constructeur appelé implicitement, soit initialisée par une instance déjà construite**.

  Ceci fait que **toute instance en C++ a un état initial connu du programmeur dans ce cas**, même si ce dernier décide de ne rien faire dans le corps d'un constructeur!
- le **passage en paramètre d'une instance par valeur** conduit à l'**initialisation du paramètre formel** par la valeur de l'argument d'appel;
- le **retour d'une instance comme valeur d'une fonction** conduit à l'**initialisation du récipient** devant contenir cette valeur par la valeur retournée;
- toute **instance flottante** comme `complexe_3 * complexe_3` ou `1.1 + 2.2 + complexe_2` est construite lorqu'elle vient de naître et détruite lorsqu'elle va mourir.

Par défaut, l'**initialisation** d'une instance se fait par une **copie membre à membre**. Comme nous le verrons au Aspects avancés du langage, il est possible de définir un **constructeur d'initialisation** de la forme:

```
une_classe :: une_classe (const une_classe & val)
```

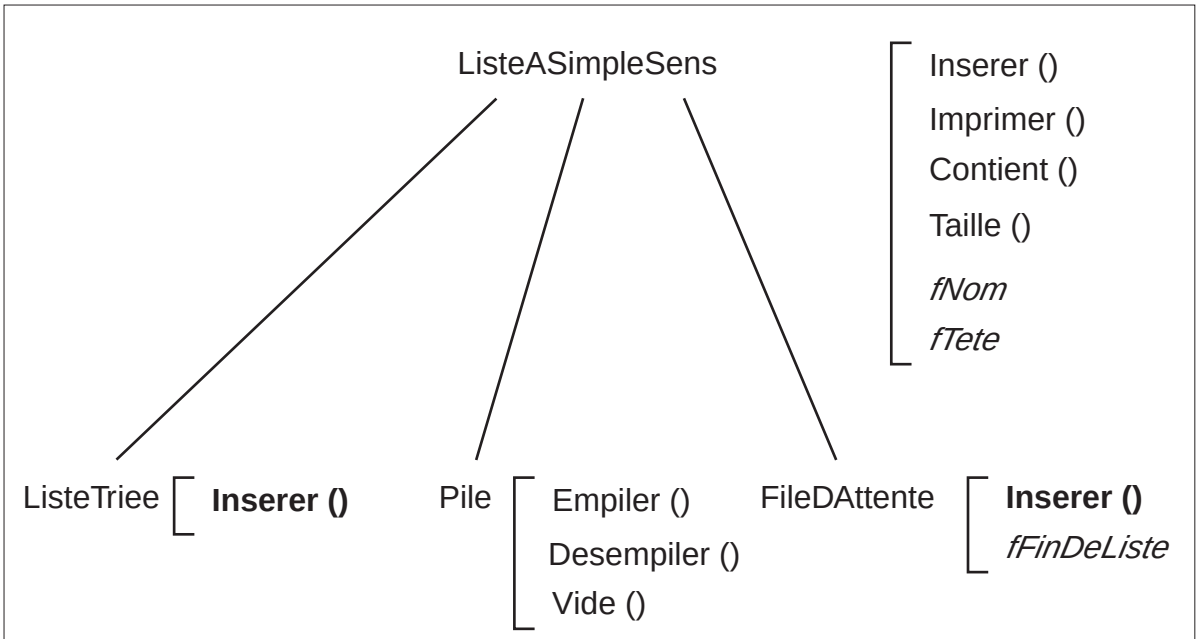
pour changer ce comportement pour les instances de `une_classe`. Ce constructeur est souvent nommé **`X (const X &)`** dans la littérature C++.

Le lecteur est invité à suivre le détail des appels et retours de fonctions, y compris de `+`, `*`, `-` et de `main`, pour assimiler cette notion de constructeurs et destructeurs appelés implicitement.

3.8 Héritage simple

Dans les deux paragraphes précédents, nous avons eu à faire à des classes isolées, sans lien entre elles. L'une des grandes forces de la programmation orientée objets est de permettre la **factorisation du code** au moyen de l'héritage. L'idée est de construire des hiérarchies de classes - de types - dont certaines sont des cas particuliers de certaines autres. Elles pourront ainsi hériter de leurs membres et fonctions membres, sans qu'on doive les spécfier à nouveau.

Pour illustrer la notion d'héritage, nous allons présenter une gestion de listes linéaires à simple sens. La hiérarchie des types pris en compte est montrée à la figure 3.1 . Dans cette figure, on a indiqué en gras les **fonctions membres re-définies**



3.1 : Hiérarchie des classes 'listes à simple sens'

dans des classes dérivées. Les termes anglais sont '*overriden member function*' et '*derived class*'. On parle aussi de **sous-classes**, *subclass* en anglais.

La signification de cette hiérarchie est entre autres qu'**une `ListeTriee` est un cas particulier de `ListeASimpleSens`**: ainsi, **toute instance de `ListeTriee` hérite** des membres `fNom` et `fTete`, ainsi que des fonctions membres `Imprimer`, `Contient` et `Taille`.

De plus, **la fonction membre `Inserer` est re-définie plus spécifiquement** pour `ListeTriee` qu'elle ne l'est pour `ListeASimpleSens`.

Il est ainsi possible de définir à volonté des hiérarchies complexes d'héritage. Celle qui est utilisée ici a une forme d'arbre, caractéristique de l'**héritage simple**: chaque classe dérivée **hérite d'une seule classe de base** (*base class* en anglais).

Nous verrons plus loin qu'il est aussi possible à une classe C++ d'avoir **plusieurs classes de base**, auquel cas on parle d'**héritage multiple**. La hiérarchie prend alors la forme d'un *graphe acyclique orienté* (DAG en anglais).

Une **proprité fondamentale de la programmation orientée objets** est que **l'héritage est strictement additif**: on peut ajouter des membres et fonctions membres dans les classes dérivées en complément à ceux qui sont hérités, on peut re-définir plus spécifiquement certaines fonctions membres, mais **on ne peut jamais faire qu'une classe dérivée soit privée de certains membres ou fonctions membres hérités**.

On peut tout au plus re-définir une fonction membre pour qu'*elle ne fasse rien*, avec un corps vide, mais elle est tout de même là!

Le fichier `Listes.h` contient les déclarations des classes formant la hiérarchie ci-dessus:

```
#ifndef __Listes__
#define __Listes__


#include <Types.h>


//   info
//   ----

typedef short      info;


//   Cellule
//   -------

struct Cellule;    //   déclaration
typedef Cellule    * CellulePtr;


struct Cellule     // définition
    {
public:

                        Cellule (info leContenu);
                        Cellule (
                             info          leContenu,
                             CellulePtr    laCelluleSuivante );


    CellulePtr          GetCelluleSuivante ()
                             { return fCelluleSuivante; }
```

```
        void                SetCelluleSuivante (
                                CellulePtr laCelluleSuivante )
                                { fCelluleSuivante = laCelluleSuivante; }

        info                GetContenu ()
                                { return fContenu; }


    private:

        info                fContenu;
        CellulePtr          fCelluleSuivante;

        };   //   Cellule


    //   ListeASimpleSens
    //   ----------------

    class ListeASimpleSens
        {
    public:

                            ListeASimpleSens (char * leNom);


        void                Inserer (info laValeur);
        void                Imprimer (char * leTitre);
        Boolean             Contient (info laValeur);
        short               Taille ();


    protected:

        char                * fNom;
        CellulePtr          fTete;


    private:

        Boolean             Contient2 (info laValeur, CellulePtr racine);

        };   //   ListeASimpleSens


    //   ListeTriee
    //   ----------

    class ListeTriee : public ListeASimpleSens
        {
    public:

                            ListeTriee (char * leNom)
                                : ListeASimpleSens (leNom)
                                // constructeur de la classe de base
                                {}
```

```
     void               Inserer (info laValeur);
          //   OVERRIDE


private:

     CellulePtr         Inserer2 (
                             info laValeur, CellulePtr laPosition );

     };   //   ListeTriee


//   Pile
//   ----

class Pile : public ListeASimpleSens
     {
public:

                        Pile (char * leNom)
                             : ListeASimpleSens (leNom)
                             {}


     void               Empiler (info laValeur);
     info               Desempiler ();
     Boolean            Vide ();

     };   //   Pile


//   FileDAttente
//   ------------

class FileDAttente : public ListeASimpleSens
     {
public:

                        FileDAttente (char * leNom);


     void               Inserer (info laValeur);
          //   OVERRIDE


protected:

     CellulePtr         fFinDeListe;

     };   //   FileDAttente


extern void             Erreur (char * leMessage);

#endif __Listes__
```

On peut remarquer plusieurs choses sur cet exemple:

- on utilise une **prédéclaration** de la classe `Cellule` avant sa **définition**, pour pouvoir déclarer le type `CellulePtr`. Il aurait été possible de s'en passer en plaçant simplement la déclaration de ce dernier dans la classe;
- les membres `fContenu` et `fCelluleSuivante` sont privés à la classe `Cellule`, mais ils peuvent être lus et modifiés par des fonctions membres publiques de la forme '**Get***membre*' et '**Set***membre*';
- la classe `ListeTriee` a une fonction membre privée `Inserer2`, appelée par la fonction membre publique `Inserer`. Cela est typique de l'**encapsulation** des détails de l'implantation d'un objet;
- les membres `fNom` et `fTete` de la classe `ListeASimpleSens` et `FileDAttente` sont **protégés**, pour que les fonctions membres des classes dérivées come `ListeTriee :: Inserer` et `Pile :: Desempiler` puissent y accéder.

  On fait de même pour `FileDAttente :: fFinDeListe`: **on laisse la porte ouverte** pour de possible classes dérivées définies par les clients du module `Listes`;
- dans le constructeur:

```
ListeTriee :: ListeTriee (char * leNom)
     : ListeASimpleSens (leNom)
     //   constructeur de la classe de base
     {}
```

  on fait appel explicitement au **constructeur de la classe de base** de `ListeTriee`. Cela est nécessaire puisque le membre `fNom` est privé, et il ne peut être initialisé autrement dans ce cas. Ensuite, le corps de ce constructeur est vide: il n'y a rien de plus à faire!

Le fichier `Listes.cp` contient les définitions de toutes les fonctions membres:

```
#include "Listes.h"

#include <StdLib.h>
#include <Stream.h>


//   Cellule
//   -------

Cellule :: Cellule (info leContenu)
     {
     fContenu = leContenu;
     fCelluleSuivante = NULL;
     }


Cellule :: Cellule (info leContenu, CellulePtr laCelluleSuivante)
     {
     fContenu = leContenu;
     fCelluleSuivante = laCelluleSuivante;
     }
```

```
//   Erreur
//   ------

void Erreur (char * leMessage)
     {
     cerr << form ("*** %s ***\n", leMessage);
     exit (1);
     }


//   ListeASimpleSens
//   ----------------

ListeASimpleSens :: ListeASimpleSens (char * leNom)
     { fNom = leNom; fTete = NULL; }


void ListeASimpleSens :: Inserer (info laValeur)
     {
     Erreur (
          "ListeASimpleSens :: Inserer"
          " doit être re-définie dans les sous-classes" );
     }


void ListeASimpleSens :: Imprimer (char * leTitre)
     {
     CellulePtr    cursor = fTete;

     cout <<
          leTitre << " " << fNom << " ( " <<
          Taille () << " elements )\n";

     while (cursor != NULL)
          {
          cout << "   " << cursor -> GetContenu () << "\n";
          cursor = cursor -> GetCelluleSuivante ();
          } //      while
     cout << "   ---\n\n";
     } //      ListeASimpleSens :: Imprimer


short ListeASimpleSens :: Taille ()
     {
     CellulePtr    curseur = fTete;
     short         aux = 0;

     while (curseur != NULL)
          {
          ++ aux;
          curseur = curseur -> GetCelluleSuivante ();
          } //      while

     return aux;
     } //      ListeASimpleSens :: Taille
```

```
Boolean ListeASimpleSens :: Contient2 (info laValeur, CellulePtr racine)
     {
     if (racine == NULL)
          return false;
     else if (racine -> GetContenu () == laValeur)
          return true;
     else
          return Contient2 (laValeur, racine -> GetCelluleSuivante ());
     }


Boolean ListeASimpleSens :: Contient (info laValeur)
     { return Contient2 (laValeur, fTete); }


//   ListeTriee
//   ----------

CellulePtr ListeTriee :: Inserer2 (info laValeur, CellulePtr laPosition)
     {
     CellulePtr    nouvelleCellule;

     if (laPosition == NULL)
          laPosition = new Cellule (laValeur);

     else
          if (laValeur <= laPosition -> GetContenu ())
               {
               //   insertion en tête de la liste
               nouvelleCellule = new Cellule (laValeur, laPosition);
               laPosition = nouvelleCellule;
               }
          else
               laPosition =
                    Inserer2 (
                         laValeur, laPosition -> GetCelluleSuivante () );

     return laPosition;
     } //     ListeTriee :: Inserer2


void ListeTriee :: Inserer (info laValeur)
     { fTete = Inserer2 (laValeur, fTete); }


//   Pile
//   ----

void Pile :: Empiler (info laValeur)
     { fTete = new Cellule (laValeur, fTete); }


info Pile :: Desempiler ()
     {
     info          res;
     CellulePtr    ancienSommet;
```

```
        if (fTete == NULL)
             Erreur (form ("Pile \"%s\" déborde par le bas", fNom));
        else
             {
             ancienSommet = fTete;
             fTete = ancienSommet -> GetCelluleSuivante ();
             res = ancienSommet -> GetContenu ();
             delete ancienSommet;
             return res;
             }
        } //      Pile :: Desempiler


    Boolean Pile :: Vide ()
        { return fTete == NULL; }


    //   FileDAttente
    //   ------------

    FileDAttente :: FileDAttente (char * leNom)
        : ListeASimpleSens (leNom)
             //   constructeur de la classe de base
        { fFinDeListe = NULL; }


    void FileDAttente :: Inserer (infolaValeur)
        {
        CellulePtr    nouvelleCellule;

        if (fFinDeListe == NULL)
             {
             fFinDeListe = new Cellule (laValeur);
             fTete = fFinDeListe;
             }
        else
             {
             //   on insère en fin de la FileDAttente
             nouvelleCellule = new Cellule (laValeur);
             fFinDeListe -> SetCelluleSuivante (nouvelleCellule);
             fFinDeListe = nouvelleCellule;
             }
        } //      FileDAttente :: Inserer
```

On peut utiliser les types constituant la hiérarchie dans le programme principal `Main.cp`, qui augmente cette hiérarchie d'une nouvelle classe `PileBornee`:

```
#include "Listes.h"

#include <Stream.h>

const short limite = 3;


//   PileBornee
//   ----------
```

```
class PileBornee : public Pile
     {
public:

                         PileBornee (char * leNom, short laCapacite);


     void                Empiler (info laValeur);
          //   OVERRIDE

     info                Desempiler ();
          //   OVERRIDE


     Boolean             Pleine ();
     short               Taille ();
          //   OVERRIDE
     short               Capacite ();


     Boolean             MemeCapaciteEtProfondeur (
                              PileBornee autrePileBornee );


private:

     short               fCapacite,     fProfondeur;

     };   //   PileBornee


PileBornee :: PileBornee (char * leNom, short laCapacite)
     : Pile (leNom)
          //   constructeur de la classe de base
     { fCapacite = laCapacite; fProfondeur = 0; }


void PileBornee :: Empiler (info laValeur)
     {
     if (fProfondeur >= fCapacite)
          Erreur (
               form (
                    "PileBornee \"%s\" déborde avec %d",
                    fNom, laValeur ));
     else
          {
          Pile :: Empiler (laValeur);       //   PAS UNE RECURSION!
          ++ fProfondeur;
          }
     } // PileBornee :: Empiler


info PileBornee :: Desempiler ()
     {
     info      res = Pile :: Desempiler ();      //   PAS UNE RECURSION!
```

```
     -- fProfondeur;
     return res;
     }


Boolean PileBornee :: Pleine ()
     { return fProfondeur == fCapacite; }


short PileBornee :: Taille ()
     { return fProfondeur; }


short PileBornee :: Capacite ()
     { return fCapacite; }


Boolean PileBornee :: MemeCapaciteEtProfondeur (
     PileBornee autrePileBornee
     )
     {
     return
          (this -> fCapacite == autrePileBornee.fCapacite)
               &&
          (this -> fProfondeur == autrePileBornee.fProfondeur);
     }


main ()
     {
     ListeTriee         gListeTriee ("Ma liste triée");
     Pile               gPile ("Ma pile");
     PileBornee         gPileBornee ("Ta pile bornee", limite + 3);
     FileDAttente       gFileDAttente ("Sa file d'attente");

     short              compteur;

     for (compteur = 1; compteur <= limite; ++ compteur)
          {
          gListeTriee.Inserer (compteur);
          gListeTriee.Inserer (17 - compteur);

          gPile.Empiler (compteur - 3);

          gFileDAttente.Inserer (compteur);
          gFileDAttente.Inserer (17 - compteur);
          } //      for


     gPile.Empiler (3);
     gPile.Empiler (5);
     gPile.Empiler (2);


     cout << "gPile.Taille () = " << gPile.Taille () << "\n";
     cout << "gPileBornee.Taille () = " << gPileBornee.Taille () << "\n";
     cout <<
          "gPile.Contient (5) = " <<
```

```
          (gPile.Contient (5) ? "vrai" : "faux")  << "\n";
     cout <<
          "gPileBornee.Contient (5) = " <<
          (gPileBornee.Contient (5) ? "vrai" : "faux") << "\n";


     while (! gPile.Vide ())
          gPileBornee.Empiler (gPile.Desempiler ());


     cout << "gPile.Taille () = " << gPile.Taille () << "\n";
     cout << "gPileBornee.Taille () = " << gPileBornee.Taille () << "\n";
     cout <<
          "gPile.Contient (5) = " <<
          (gPile.Contient (5) ? "vrai" : "faux")  << "\n";
     cout <<
          "gPileBornee.Contient (5) = " <<
          (gPileBornee.Contient (5) ? "vrai" : "faux") << "\n";
     cout << "\n";


     gListeTriee.Imprimer ("-->");
     gPile.Imprimer ("-->");
     gPileBornee.Imprimer ("-->");
     gFileDAttente.Imprimer ("-->");

     cout <<
          "gPileBornee.Capacite () = " <<
          gPileBornee.Capacite () << "\n";

     } //      main
```

Les résultats de ce programme sont:

```
gPile.Taille () = 6
gPileBornee.Taille () = 0
gPile.Contient (5) = vrai
gPileBornee.Contient (5) = faux
gPile.Taille () = 0
gPileBornee.Taille () = 6
gPile.Contient (5) = faux
gPileBornee.Contient (5) = vrai


--> Ma liste triée ( 3 elements )
   14
   15
   16
   ---


--> Ma pile ( 0 elements )
   ---
```

```
--> Ta pile bornee ( 6 elements )
   -2
   -1
   0
   3
   5
   2
   ---


--> Sa file d'attente ( 6 elements )
   1
   16
   2
   15
   3
   14
   ---

gPileBornee.Capacite () = 6
```

Nous appelons **programmation chirurgicale** le style d'écriture propre aux objets: on intervient de manière très précise sur un point très particulier de l'ensemble, comme dans:

```
Boolean Pile :: Vide ()
     { return fTete == NULL; }
```

3.9 Polymorphisme et fonctions membres virtuelles

On a vu qu'une sous-classe d'une classe donnée définit un cas plus particulier de cette dernière. Ainsi, dans:

```
class humain
     {
public:
     int            annee_de_naissance;
     // … … … … … … …
     };

class femme: public humain
     {
public:
     int            nb_enfants;
     // … … … … … … …
     };

class homme: public humain
     {
public:
     Boolean        barbu;
     // … … … … … … …
     };
```

les types `femme` et `homme` sont *des types d'humains particuliers*.

Une question importante est: à quelle condition des variables de l'un de ces types peuvent-elles recevoir par effectation une valeur d'un autre type? On appelle **polymorphisme** (plusieurs formes) le fait qu'**une variable peut être déclarée d'un type et contenir une valeur d'un autre type**.

La réponse à la question précédente est:

toute variable pointeur d'un type `'une_classe *'`
peut prendre comme valeur
un pointeur d'un type `'descencant_de_une_classe *'`

Rappelons qu'un **descendant** d'une classe est soit une de ses classes dérivées, soit une classe dérivée d'une classe dérivée, et ainsi de suite vers les feuilles de la hiérarchie des classes ayant comme racine la classe considérée.

Le polymorphisme n'a de sens que pour les pointeurs sur des instances, soit dans le cas d'objets alloués *dynamiquement*. Le pointeur est nécessaire car un objet non alloué dynamiquement comme:

```
humain    un_humain;
```

ne peut contenir un `homme` ou une `femme`: les instances de des deux types ont des *membres supplémentaires* par rapport à `humain` et occupent plus de place en mémoire. Le problème de la *gestion de variantes* se pose dans ce cas de manière incontournable!

Ainsi, étant données les déclarations complémentaires:

```
class femme_rousse: public femme
     {
public:
     char           * couleur_des_yeux;
     // … … … … … … …
     };


typedef humain          * humain_ptr;
typedef femme           * femme_ptr;
typedef homme           * homme_ptr;
typedef femme_rousse    * femme_rousse_ptr;


humain_ptr              un_humain_ptr;
femme_ptr               une_femme_ptr;
homme_ptr               un_homme_ptr;
femme_rousse_ptr        une_femme_rousse_ptr;
```

on a la situation suivante:

- les affectations:

  ```
  un_humain_ptr = une_femme_ptr;
  un_humain_ptr = un_homme_ptr;
  un_humain_ptr = une_femme_rousse_ptr;
  ```

  sont **toujours - statiquement, à la compilation - sémantiquement correctes**. Après l'une quelconque de ces affectations, la variable

`un_humain_ptr`, qui reste bien entendu du type `humain_ptr`, contient une valeur d'un type pointeur sur un descendant de `humain`;

- les affectations:

```
un_humain_ptr = 0;
```

et:

```
un_humain_ptr = NULL;
```

sont **sémantiquement correctes**, toutefois elle permettent d'accéder ensuite par la notation '`* un_humain_ptr`' à un **objet inexistant**;

- l'affectation:

```
une_femme_ptr = un_humain_ptr;    // ILLEGAL
```

est **sémantiquement incorrecte**, car elle enfreint la règle du polymorphisme énoncée ci-dessus: `humain` n'est pas une classes descendant de `femme`.

En cas de polymorphisme, comment fonctionne l'héritage des procédures et fonctions? Plus précisément, **quel code sera exécuté** en réponse à un message correspondant à une fonction membre qui est re-définie à différents niveaux dans la hiérarchie des classes?

Soient les déclarations:

```
classe une_classe
     {
public:
     // différentes fonctions membres…
     //   … … …
     };

typedef une_classe * une_classe_ptr;

une_classe_ptr     pointeur_sur_une_instance = …;
```

Lorsqu'on exécute:

```
pointeur_sur_une_instance -> une_fonction_membre ()
```

on cherche de bas en haut dans la hiérarchie des classes une version de `une_fonction_membre` pour l'exécuter. Si aucune telle définition n'est trouvée, il y a une erreur sémantique statique, dépistée à la compilation.

Le **point de départ de cette recherche dépend du fait que** `une_fonction_membre` **est spécifiée `virtual` ou non:**

- si `une_fonction_membre` n'est **pas** spécifiée `virtual`, **la déclaration de `pointeur_sur_une_instance` est déterminante**: la recherche d'une version de `une_fonction_membre` commence **dans la classe `une_classe`**;
- si `une_fonction_membre` est spécifiée `virtual`, **la valeur de `pointeur_sur_une_instance` est déterminante**: la recherche d'une version de `une_fonction_membre` commence **dans la classe de `* pointeur_sur_une_instance`**, qui peut être `une_classe` ou n'importe laquelle de ses descendantes.

3.10 Polymorphisme: exemple

Voici un exemple illustrant le polymorphisme:

```
#include <types.h>
#include <stream.h>

class humain
     {
public:

                    humain (int annee)          //   un constructeur
                         { annee_de_naissance = annee; }

                    humain ()          //   LE constructeur par défaut
                         { annee_de_naissance = 1956; }

     virtual char  * ecrire ()
          // on peut jouer sur l'absence de ce 'virtual'
                         {
                         return
                              form ( "humain (%d)", annee_de_naissance );
                         }


private:

     int            annee_de_naissance;

     }; // humain


class femme: public humain
     {
public:


                    femme (int annee, int enfants)
                         //   un constructeur
                         : humain (annee),
                         //   constructeur de la super-classe
                         nb_enfants (enfants)
                         //   liste d'initialisation
                         { /* ce corps est vide */ }


     char           * ecrire ()
                         {
                         return
                              form (
                                   "femme (%d, %d)",
                                   annee_de_naissance, nb_enfants );
                         }


private:
```

```
     int            nb_enfants;

     }; // femme


char * out_bool (Boolean le_booleen)
     { return (le_booleen == 0) ? "faux" : "vrai"; }


class homme: public humain
     {
public:
                    homme (int, Boolean);   //   un constructeur

     char           * ecrire ()
                         {
                         return
                              form (
                                   "homme (%d, %s)",
                                   annee_de_naissance, out_bool (barbu));
                         }

private:

     Boolean        barbu;

     }; // homme


homme :: homme (int annee, Boolean barb)
     : humain (annee)        //   constructeur de la super-classe
                             //   "humain" est ici superflu
     { barbu = barb; }


main ()
     {
     femme          marie (1953, 3);
     homme          jean (1954, true);
     humain         un_humain;
     humain         * quidam = & un_humain;

     cout << "marie initiale = " << marie.ecrire () << "\n";
     cout << "jean initial = " << jean.ecrire () << "\n";
     cout << "un_humain = " << un_humain.ecrire () << "\n";
     cout << "quidam initial = " << quidam -> ecrire () << "\n\n";

     marie.nb_enfants ++;
     jean.barbu = ! jean.barbu;

     cout << "marie finale = " << marie.ecrire () << "\n";
     cout << "jean final = " << jean.ecrire (); << "\n";

     quidam = & marie;

     cout << "quidam final = " << quidam -> ecrire () << "\n";
     }
```

Les résultats produits par ce programme **avec la spécification `virtual`** pour la fonction membre `humain :: ecrire` sont:

```
marie initiale = femme (1953, 3)
jean initial = homme (1954, vrai)
un_humain = humain (1956)
quidam initial = humain (1956)


marie finale = femme (1953, 4)
jean final = homme (1954, faux)
quidam final = femme (1953, 4)
```

car l'envoi du message `quidam -> ecrire ()` déclenche dans ce cas l'exécution de la fonction membre **`femme :: ecrire`**.

Sans la spécification `virtual` pour la fonction membre `humain :: ecrire`, les résultats deviennent:

```
marie initiale = femme (1953, 3)
jean initial = homme (1954, vrai)
un_humain = humain (1956)
quidam initial = humain (1956)


marie finale = femme (1953, 4)
jean final = homme (1954, faux)
quidam final = humain (1953)
```

parce que l'envoi du message `quidam -> ecrire ()` déclenche l'exécution de la fonction **`humain :: ecrire`**, **indépendamment du type de de la valeur** de `quidam`, qui est '`femme *`'.

Rappelons que c'est la **déclaration du pointeur** qui est utilisée pour déterminer quelle fonction membre sera appelée en réponse à un message correspondant à une fonction membre **non `virtual`**, et que c'est la **valeur du pointeur** qui est utilisée dans le cas **`virtual`**.

Bien entendu, le type du pointeur affecté à `quidam` dans notre exemple ne peut être **qu'un pointeur sur une instance de la classe `humain` ou de l'une de ses descendantes**.

Le constructeur `femme :: femme` dans:

```
class femme: public humain
     {
public:

                    femme (int annee, int enfants)
                         //   un constructeur
                         : humain (annee),
                         //   constructeur de la super-classe
                         nb_enfants (enfants)
                         //   liste d'initialisation
                         { /* ce corps est vide */ }
```

utilise une **liste d'initialisation**, d'apparence analogue aux appels aux constructeurs de la classe de base, mais où ce sont les *noms des membres* de la classe elle-même qui apparaissent avant les parenthèses.

Dans ce cas, cela est une alternative à l'affectation explicite:

```
femme (int annee, int enfants)
     //   un constructeur
     : humain (annee),
     //   constructeur de la super-classe
     { nb_enfants = enfants; }
```

Nous verrons au Aspects avancés du langage que les **membres d'une classe qui sont eux-mêmes des instances** doivent être initialisés par ce moyen dans les constructeurs.

3.11 Classes abstraites

Le lecteur aura remarqué que dans l'exemple du paragraphe 3.8, page 24, il n'est pas possible de manipuler une `ListeASimpleSens` en tant que telle: toute tentative d'insertion donne lieu à message d'erreur, suivi de l'arrêt du programme. C'est en fait un choix que nous avons fait: il serait possible d'insérer des éléments dans une `ListeASimpleSens`, par exemple en fin de liste systématiquement.

Il existe un moyen d' **empêcher de créer des instances** de `ListeASimpleSens` **sans devoir attendre l'exécution** du programme pour le vérifier. Nous allons spécifier pour cela dans une seconde version que la classe `ListeASimpleSens` est une **classe abstraite**: elle n'est qu'un **modèle pour des classes dérivées**, mais n'est **pas destinée à la création d'instances** de son propre type.

En C++, une classe est abstraite dès qu'elle a au moins une **fonction membre virtuelle pure**, dont la déclaration contient '`= 0`' à la place du corps.

Notre exemple devient:

```
class ListeASimpleSens       //   CLASSE ABSTRAITE
     {
public:
     //   … … …
     virtual void  Inserer (info laValeur) = 0;
          //   VIRTUELLE PURE
```

et l'on doit bien sûr éliminer du fichier `Listes.cp` l'ancienne définition de `ListeASimpleSens :: Inserer`.

Pour pouvoir instancier une sous-classe d'une classe abstraite, il est nécessaire de fournir une **version concrète** de toutes les fonctions membres virtuelles pures. Ainsi, dans le cas de la classe `Pile`, nous devons définir `Pile :: Inserer`, par exemple par:

```
class Pile : public ListeASimpleSens
     {
public:
     //   … … …
     void          Inserer (info laValeur)
          //   OVERRIDE
                        { this -> Empiler (laValeur); }
```

☞ Comme `PileBornee` est une classe dérivée de `Pile`, elle est instanciable dès lors que `Pile` l'est: il n'est pas nécessaire de fournir une version concrète de `Inserer` pour elle, puisque la **version concrète** ci-dessus est **héritée**.

3.12 Ré-utilisation de code

Pour illustrer la **factorisation** de caractéristiques communes à plusieurs types et la **gestion de variantes** dans les types, voici l'exemple de la gestion d'un **arbre de recherche** dans lequel nous allons **abstraire** la gestion de l'arbre de recherche des détails concernant le type des valeurs à stocker et des opérations de comparaison et d'impression de ces valeurs.

Le but est de pouvoir ré-utiliser toute la spécification de l'arbre de recherche pour y **insérer des valeurs de types variés**, **sans modification ni recompilation du module `arbre_de_recherche`!**

Spécifions que:

- un **`noeud`** contient deux champs qui sont eux-mêmes des pointeurs sur des instances de `noeud`;
- il est possible de **comparer** deux valeurs du type `noeud` par la fonction membre booléenne **`noeud`** :: **`plus_petit_que`**;
- il est possible d'**imprimer** une valeur du type `noeud` sur une ligne au moyen de la fonction membre **`noeud`** :: **`ecrire_valeur`**.

Le fait que l'on doive **comparer et imprimer les valeurs des nœuds et non les valeurs stockées dans ces nœuds** est imposé précisément par le fait que nous voulons avoir une certaine flexibilité sur le type des valeurs stockées dans les nœuds. Nous ne pouvons donc pas stocker la valeur dans le type `noeud` tout en lui permettant d'être de types variés selon les besoins.

☞ En fait, **nous ne stockons tout simplement pas la valeur présente au nœud dans le type `noeud`**!

Les fonctions membres qui manipulent ces `noeud` sont déclarées *virtuelles pures*, faisant de noeud une *classe abstraite*. Ainsi, dans:

```
virtual
     Boolean plus_petit_que (noeud_ptr autre_noeud) = 0;
virtual
     void ecrire_valeur () = 0;
```

de l'exemple ci-dessous, *on ne peut pas créer d'instances de* `noeud`, *mais seulement de ses sous-classes*. Comme indiqué au paragraphe précédent, les fonctions membres virtuelles pures héritées doivent être définies pour les sous-classes '*noeud concret*' que nous voudrons insérer dans nos arbres de recherche.

La ré-utilisation de code est réalisée dans cet exemple en définissant **des cas plus particuliers de `noeud`**, contenant une valeur d'une type adéquat comme un entier ou

un flottant. Le module `ArbreDeRecherche` fournit les services déclarés dans le fichier **`ArbreDeRecherche.h`**, dont le contenu est:

```
#include <types.h>
//   #include <stream.h>

class noeud   //   classe abstraite
     {
     typedef noeud      * noeud_ptr;
     friend class       arbre_de_recherche;


public:

                        noeud ()      //   le constructeur par defaut
                             {
                             sous_arbre_gauche = sous_arbre_droit =
NULL;
                             }


     virtual Boolean    plus_petit_que (noeud_ptr autre_noeud) = 0;
          // virtuelle pure


     virtual void       ecrire_valeur () = 0;
          // virtuelle pure


private:

     noeud_ptr          sous_arbre_gauche, sous_arbre_droit;

     };   //   noeud


class arbre_de_recherche
     {
public:

                        arbre_de_recherche ();

     void               inserer (noeud_ptr le_noeud);
     void               imprimer ();


private:

     noeud_ptr          racine;

     void               inserer2 (
                             noeud_ptr & l_arbre, noeud_ptr le_noeud );
     void               imprimer2 (noeud_ptr l_arbre);

     }; // arbre_de_recherche
```

Le fichier **ArbreDeRecherche.cp**, quant à lui, contient les définitions suivantes:

```
#include "ArbreDeRecherche.h"

arbre_de_recherche :: arbre_de_recherche ()
     { racine = NULL; }


void arbre_de_recherche :: imprimer ()
     { imprimer2 (racine); }


void arbre_de_recherche :: inserer (noeud_ptr le_noeud)
     { inserer2 (racine, le_noeud); }


void arbre_de_recherche :: imprimer2 (noeud_ptr l_arbre)
     {
     if (l_arbre != NULL)
          {
          imprimer2 ( l_arbre -> sous_arbre_gauche );

          l_arbre -> ecrire_valeur ();

          imprimer2 ( l_arbre -> sous_arbre_droit );
          }
     }


void arbre_de_recherche :: inserer2 (
     noeud_ptr & l_arbre, noeud_ptr le_noeud
     )
     {
     if (l_arbre != NULL)
          if (! l_arbre -> plus_petit_que (le_noeud))
               inserer2 (l_arbre -> sous_arbre_gauche, le_noeud);
          else
               inserer2 (l_arbre -> sous_arbre_droit, le_noeud);

     else
          l_arbre = le_noeud;
     } // arbre_de_recherche :: inserer2
```

On peut alors ré-utiliser le code gérant un arbre de recherche dans le programme **Main.cp** suivant:

```
#include "ArbreDeRecherche.h"

#include <stream.h>
#include <StdLib.h>     // pour "rand ()" et "RAND_MAX"

float random ()
     { return float (rand ()) / RAND_MAX; }     // conversion explicite

class noeud_reel: public noeud
     {
     typedef noeud_reel      * noeud_reel_ptr;
```

```
public:

                        noeud_reel (float la_valeur)
                             // un constructeur

                             : noeud ()
                             // constructeur de la super-classe
                             // superflu car il n'a pas de parametres

                             {
                             val = la_valeur;
                             // liste d'initialisation possible ici
                             }


     // les deux occurences de 'virtual' ci-dessous sont superflues

     virtual Boolean    plus_petit_que (noeud_ptr autre_noeud);
          //   OVERRIDE

     virtual void       ecrire_valeur ();
          //   OVERRIDE


private:

     float              val;

     }; // noeud_reel


Boolean noeud_reel :: plus_petit_que (noeud_ptr autre_noeud)
     { return val < noeud_reel_ptr (autre_noeud) -> val; }


void noeud_reel :: ecrire_valeur ()
     { cout << "   " << val << "\n"; }


void demo_reels ()
     {
     const int              taille = 5;
     arbre_de_recherche     mon_pommier;

     cout << "Valeurs aléatoires obtenues:" << "\n";

     for ( int i = 1; i <= taille; i++ )
          {
          noeud_ptr la_valeur = new noeud_reel ( random () );
          la_valeur -> ecrire_valeur ();
          mon_pommier.inserer ( la_valeur );
          }

     cout << "\n" << "Les mêmes triées:" << "\n";
     mon_pommier.imprimer ();
     }    //   demo_reels
```

```
class noeud_chaine: public noeud
     {
     typedef noeud_chaine    * noeud_chaine_ptr;

public:
                         noeud_chaine (char * la_valeur)
                              // un constructeur

                              : noeud ()
                              // constructeur de la super-classe

                              { val = la_valeur; }


     Boolean             plus_petit_que (noeud_ptr autre_noeud);
     void                ecrire_valeur ();


private:

     char                * val;

     }; // noeud_chaine


Boolean noeud_chaine :: plus_petit_que (noeud_ptr autre_noeud)
     {
     return
          strcmp (val, noeud_chaine_ptr (autre_noeud) -> val) < 0;
     }


void noeud_chaine :: ecrire_valeur ()
     { cout << "   " << val << "\n"; }


noeud_chaine_ptr creer_noeud_chaine (char * la_chaine)
     {
     cout << "   " << la_chaine << "\n";
     return new noeud_chaine (la_chaine);
     }


void demo_chaines ()
     {
     arbre_de_recherche      mon_citronnier;

     cout << "Quelques noms de villes:" << "\n";
     mon_citronnier.inserer ( creer_noeud_chaine ("Lausanne") );
     mon_citronnier.inserer ( creer_noeud_chaine ("Genève") );
     mon_citronnier.inserer ( creer_noeud_chaine ("Pise") );

     cout << "\n" << "Les mêmes triés:" << "\n";
     mon_citronnier.imprimer ();
     }    //   demo_chaines
```

```
main ()
     {
      demo_reels ();
      cout << "\n";
      demo_chaines ();
     }
```

Les résultats produits par ce programme sont:

```
Valeurs aléatoires obtenues:
   0.513871
   0.175726
   0.308634
   0.534532
   0.94763


Les mêmes triées:
   0.175726
   0.308634
   0.513871
   0.534532
   0.94763


Quelques noms de villes:
   Lausanne
   Genève
   Pise


Les mêmes triés:
   Genève
   Lausanne
   Pise
```

La classe `noeud_reel` définit de manière spécifique les fonctions membres:

```
Boolean noeud_reel :: plus_petit_que (noeud_ptr autre_noeud)
     { return( val < noeud_reel_ptr (autre_noeud) -> val); }
```

et:

```
void noeud_reel :: ecrire_valeur ()
     { cout << "   " << val << "\n"; }
```

La **conversion de type** (coercion ou **type cast** en anglais) est nécessaire ici. En effet:

- `noeud :: plus_petit_que` a un paramètre du type `noeud_ptr`, passé par valeur. Il est nécessaire de re-définir cette fonction avec le même profil que celle qui est héritée, pour qu'elle soit appelée par *polymorphisme*. Sans cela, on aurait un avertissement du compilateur, parce qu'*on masquerait la fonction héritée*, mais sans la re-définir localement!

- la comparaison de deux valeurs n'a de sens qu'entre valeurs du même type. Le receveur du message `plus_petit_que` étant du type `noeud_reel`, on doit convertir l'`autre_noeud` à ce même type.

Remarquons encore que dans la fonction `random`, on a dû faire une conversion de type explicite de `rand ()` au type réel `float`, pour que le type du résultat de la division soit aussi réel. Sans cela, notre générateur `aléatoire` effectuerait une division entière, et retournerait toujours la valeur 0! Cela donne:

```
float random ()
     { return float (rand ()) / RAND_MAX; }
```

Nous verrons en détail au Aspects avancés du langage les possibilités de conversions offertes par C++, dont les **constructeurs de conversion** de la forme:

```
type_destination ::
     type_destination (type_source valeur_a_convertir)
```

et les **fonctions membres de conversion** de la forme:

```
type_source :: operator type_destination ()
```

Nous aurions pu déclarer la classe `noeud` **locale** à `arbre_de_recherche`, mais *cela n'apporte rien de particulier*: tout se passe comme si elle était déclarée au même niveau que `arbre_de_recherche`. En particulier, la visibilité de cette classe ne serait pas restreinte aux seules fonctions membres de la classe `arbre_de_recherche` dont elle est une déclaration *privée*. D'ailleurs nous avons pu utiliser l'identificateur `noeud_ptr` dans la fonction `main`, alors qu'il est défini dans la classe `noeud`.

3.13 Membres et fonctions membres statiques

Tous les membres de classes que nous avons rencontrés jusqu'ici étaient des **membres d'instances**, dont il existe un exemplaire dans chaque instance du type considéré. Ainsi, dans le type `humain` déclaré par:

```
class humain
     {
     //   … … …
     int                 annee_de_naissance;
     //   … … …
     };
```

chaque instance de `humain` contient son propre exemplaire du membre `annee_de_naissance`.

Il est possible en C++ de définir des **membres de classe** ou **membres statiques**, qui n'existent qu'en un seul exemplaire quel que soit le nombre d'instances créées, et qui sont en fait **propres à la classe**.

Ce sont des **variables globales dont la visibilité est restreinte à l'intérieur de la définition de la classe et à l'aide de l'opérateur d'accès à un niveau de déclaration** '`::`'. On indique donc ce fait par le mot clé `static` précédant un membre de classe ou de structure, d'où les termes de *membre statique* et *fonction membre statique* employés en C++.

Dans l'exemple:

```
#include <types.h>
#include <stream.h>

class cheval
     {
public:
     int                annee_de_naissance;
     char               * couleur;

     static int         nb_instances;          // déclaration

                        cheval (int annee, char * kouleur)

                             //   liste d'initialisation
                             :    annee_de_naissance (annee),
                                  couleur (kouleur)

                             { ++ nb_instances; }

     }; // cheval


//   définition (et donc initialisation) du membre statique

int       cheval :: nb_instances = 0;


main ()
     {
     const     int taille = 5;

     for ( int i = 1; i <= taille; ++ i )
          {
          cheval * le_cheval =
               new cheval ( 1980 + i, "brun pommelé" );
          }

     cout << form (
          "Nombre d'instances de 'cheval' construites: %d",
          cheval :: nb_instances
          );
     }
```

on utilise le membre statique `nb_instances` pour compter le nombre d'instances différentes de la classe `cheval` ayant été construites.

La notation `cheval :: nb_instances` permet d'accéder à cette variable: une telle notation est nécessaire, car **un membre statique n'appartient à aucune instance particulière d'aucune classe**.

On notera que `static` pour un membre de classe **ne fait que déclarer l'identificateur** correspondant, et que l'**on doit le définir en dehors de la classe** au moyen de:

```
int cheval :: nb_instances = 0;       // définition
```

afin de lui *allouer de la place en mémoire*. Ceci met en évidence le fait qu'une variable de classe est fondementalement une **variable globale à visibilité restreinte**.

Si l'on omet cette définition de la variable de classe, qui doit correspondre pour ce qui est du type à la déclaration qui en est faite dans la classe elle-même, on obtient *un message à l'édition de liens seulement*, et non à la compilation, puisqu'elle peut se trouver dans un autre fichier source.

Le résultat produit par le programme ci-dessus est:

```
Nombre d'instances de 'cheval' construites: 5
```

Rappelons que l'opérateur '`::`' d'accès à un niveau de déclaration permet aussi de **contourner le masquage d'un identificateur** qui est effectué par une déclaration imbriquée du même identificateur, comme on l'a vu au paragraphe 2.9, page 16.

3.14 Pointeurs sur des fonctions membres

L'exemple de la gestion de l'arbre de recherches du paragraphe 3.12, page 42 peut se prêter à une factorisation du code encore plus poussée. Il s'agit dans ce cas d'**abstraire la notion de traversée de l'arbre**, dans laquelle on visite une fois chaque nœud pour y appliquer une certaine opération.

☞ Vue sous cet angle, la fonction membre `arbre_de_recherche :: imprimer` n'est *qu'une traversée particulière*, dans laquelle l'opération à appliquer sur chaque nœud est `ecrire_valeur`.

Il est important que les fonctions qui seront fournies comme paramètres à `arbre_de_recherche :: traverser` soient des fonctions membres de `noeud`, pour qu'elles puissent s'appliquer à des nœuds de l'arbre.

Nous allons donc définir un type pointeur sur une fonction membre de la classe `noeud`, sous la forme:

```
typedef void  (noeud :: * pointeur_sur_fonction_membre) ();
```

ainsi que déclarer les fonctions membres `traverser` et `traverser2` dans la classe `arbre_de_recherche`:

```
class arbre_de_recherche
     {
public:

                        arbre_de_recherche ();  //   un constructeur


     void               inserer (noeud_ptr le_noeud);
     void               traverser (
                             pointeur_sur_fonction_membre funct );
     void               imprimer ();
```

```
private:

     noeud_ptr          racine;
     void               inserer2 (
                             noeud_ptr & l_arbre, noeud_ptr le_noeud);
     void               traverser2 (
                             noeud_ptr                       l_arbre,
                             pointeur_sur_fonction_membre    funct );

     }; // arbre_de_recherche
```

Ces deux fonctions membres sont définies respectivement par:

```
void arbre_de_recherche :: traverser (
     pointeur_sur_fonction_membre funct
     )
     { traverser2 (racine, funct); }
```

et:

```
void arbre_de_recherche :: traverser2 (
     noeud_ptr                       l_arbre,
     pointeur_sur_fonction_membre    funct
     )
     {
     if (l_arbre != NULL)
          {
          traverser2 ( l_arbre -> sous_arbre_gauche, funct );

          (l_arbre ->* funct) ();

          traverser2 ( l_arbre -> sous_arbre_droit, funct );
          }
     } // arbre_de_recherche :: traverser2
```

On peut alors définir:

```
void arbre_de_recherche :: imprimer ()
     { traverser (& noeud :: ecrire_valeur); }
```

On notera que dans l'écriture:

```
(l_arbre ->* funct) ();
```

on a en fait l'envoi du message 'fonction membre pointée par `funct`' à l'instance pointée par `l_arbre`, ce qui explique la double indirection sémantique contenue dans l'opérateur `->*`, qui s'écrit *sans espaces intermédiaires*.

Cet envoi de message n'a pas de paramètres, d'où les parenthèses sans rien entre elles:

```
(l_arbre ->* funct) ();
```

Le 'défaut' de ce genre de traversée est qu'elle est faite *toute en une fois*, sans qu'on puisse par exemple l'arrêter en cours de route dès qu'un élément de l'arbre possédant une certaine caractéristique est atteint. Nous verrons au Aspects avancés du langage

que la notion d'**itérateur sur une structure de données** généralise cette notion de traversée, tout en lui donnant la souplesse qui manque à celle de cet exemple.

3.15 Héritage multiple

Les exemples d'héritage que nous avons vus jusqu'à présent ont toujours été basés sur un raffinement d'*une* classe pré-existante. Ce type d'**héritage simple**, où une sous-classe a une seule superclasse, conduit à des *hiérarchies de classes* ayant la forme d'un **arbre** dont la racine est une classe n'ayant pas de superclasse. On peut avoir autant de tels arbres que l'on veut, les relations de classe à super-classe se trouvant toujours confinées à un seul arbre.

Lorsqu'on passe d'une classe à ses sous-classes dans une hiérarchie d'héritage simple, on a une relation **OU exclusif**: dans l'exemple des humains que nous avons illustré au paragraphe 3.9, page 35, **un humain plus spécifique que le type général est soit une femme, soit un homme**, mais pas les deux à la fois.

Qu'en est-il si nous désirons modéliser un type d'objets qui soit un *raffinement de plusieurs classes*, et non pas seulement d'une seule? Par exemple, nous pouvons définir, en complément aux types `humain`, `femme` et `homme`, les types `femme_mariee` et `homme_veuf`. Ce mécanisme, qui n'est pas disponible dans tous les langages orientés objets, s'appelle l'**héritage multiple**.

On peut objecter que nous pourrions aussi définir des membres `annee_de_mariage` et `annee_de_veuvage` dans le type `humain`, plutôt que des sous-types, mais cela nous empêcherait de manipuler des instances du type `homme_veuf` en tant que telles, par exemple.

Un autre argument en faveur de l'héritage multiple est fourni par des considérations de **ré-utilisation de code**: si les types `humain`, `femme` et `homme` sont placés en librarie, il est possible, **sans avoir accès à leur code source**, d'en définir des sous-classes comme `femme_rousse` et `homme_brun`.

Par contre, il nous est impossible de les modifier pour y rajouter des membres comme `annee_de_mariage` et `annee_de_veuvage`. Nous pourrions définir une sous-classe `humain_plus_complet` de `humain` avec ces champs, mais comment faire pour que `femme` et `homme` en héritent aussi, sinon en définissant aussi des sous-classes `femme_plus_complete` et `homme_plus_complet`? En un mot, cela nous conduirait à construire toute une nouvelle hiérarchie de classes, *de structure parallèle à celle qui est placée en librairie*, dont une partie seulement serait exploitée dans ce cas, à savoir le seul type `humain`.

En permettant de **spécifier qu'un `homme_veuf` est un homme ET qu'il est veuf**, l'héritage multiple apporte une solution élégante à ce problème. Voici un exemple en C++ illustrant ce phénomène:

```
#include <types.h>
#include <stream.h>

class humain
     {
public:
```

```
                        humain (int annee = 1956)
                             { annee_de_naissance = annee; }


     virtual void       ecrire ()
                             {
                              cout <<
                                  form (
                                       "humain (%d)",
                                       annee_de_naissance );
                             }

private:

     int                annee_de_naissance;

     }; // humain


class femme: virtual public humain
     {
public:
                        femme (int annee, int enfants)
                             //   un constructeur

                             :    humain(annee)
                             //   constructeur de la superclasse

                             { nb_enfants = enfants; }

     virtual void       ecrire ()
                             {
                             cout << form (
                                  "femme (%d, %d)",
                                  annee_de_naissance, nb_enfants
                                  );
                             }

private:

     int                nb_enfants;

     }; // femme


char * out_bool ( Boolean le_booleen )
     { return( (le_booleen == 0) ? "faux" : "vrai" ); }


class homme: virtual public humain
     {
public:
                        homme (int annee, Boolean barb)
                             //   un constructeur

                             :    (annee)
                             //   constructeur de la superclasse
```

```
                              { barbu = barb; }

     virtual void       ecrire ()
                              {
                              cout << form (
                                   "homme (%d, %s)",
                                   annee_de_naissance, out_bool(barbu)
                                   );
                              }

private:

     Boolean            barbu;

     }; // homme


class marie_ou_mariee: virtual public humain
     {
public:
                         marie_ou_mariee (int mariage)
                              // un constructeur
                              { annee_de_mariage = mariage; }

     virtual void       ecrire ()
                              {
                              cout << form (
                                   "marie_ou_mariee (%d, %d)",
                                   annee_de_naissance, annee_de_mariage
                                   );
                              }

private:

     int                annee_de_mariage;

     }; //      marie_ou_mariee


class veuf_ou_veuve: virtual public humain
     {
public:
                         veuf_ou_veuve (int veuvage)
                              // un constructeur
                              { annee_de_veuvage = veuvage; }

     virtual void       ecrire ()
                              {
                              cout << form(
                                   "veuf_ou_veuve (%d, %d)",
                                   annee_de_naissance, annee_de_veuvage
                                   );
                              }

private:

     int                annee_de_veuvage;
```

```
    }; // veuf_ou_veuve


class femme_mariee: public femme, public marie_ou_mariee
    {
public:

                        femme_mariee (
                            int naissance, int mariage, int enfants )
                            //   un constructeur

                            : // constructeurs des deux superclasses
                            femme (naissance, enfants),
                            marie_ou_mariee (mariage)

                            { /* Ce corps est vide */ }

    virtual void        ecrire ()
                            {
                            cout << form (
                                "femme_mariee (%d, %d, %d)",
                                annee_de_naissance,
                                annee_de_mariage,
                                nb_enfants
                                );
                            }

    }; // femme_mariee


class homme_veuf: public homme, public veuf_ou_veuve
    {
public:

                        homme_veuf (
                            int naissance, int veuvage, Boolean barb )
                            // un constructeur

                            : // constructeurs des deux superclasses
                            homme (naissance, barb),
                            veuf_ou_veuve (veuvage)

                            { /* Ce corps est vide */ }

    virtual void        ecrire ()
                            {
                            cout << form (
                                "homme_veuf (%d, %d, %d)",
                                annee_de_naissance,
                                annee_de_veuvage,
                                out_bool (barbu)
                                );
                            }

    }; // homme_veuf
```

```
main ()
     {
     femme_mariee  marie (1953, 1975, 3);
     homme_veuf    jean (1954, 1988, true);

     cout << "marie = "; marie.ecrire (); cout << "\n";
     cout << "jean = "; jean.ecrire (); cout << "\n";
     }
```

Ce programme fournit comme résultat:

```
marie = femme_mariee (1956, 1975, 3)
jean = homme_veuf (1956, 1988, vrai)
```

On notera la déclaration:

```
class femme_mariee: public femme, public marie_ou_mariee
```

indiquant que `femme_mariee` hérite des deux classes `femme` et `marie_ou_mariee`. Le mot clé `public` indique ici la visibilité des champs de la superclasse dans la classe que l'on définit:

- **`public`** signifie que les champs `public` et `protected` de la super-classe restent `public` et `protected`, respectivement, dans la nouvelle classe;
- **`private`** signifie que les champs `public` et `protected` de la super-classe sont des champs `private` de la nouvelle classe, et ne sont donc visibles que dans cette seule classe, à l'exception de ses propres sous-classes et du niveau de déclaration la contenant.

`private` *coupe donc la visibilité des* `public` *et* `protected` *hérités vers le bas de la hiérarchie des classes*.

Le mot clé **`protected`** ne peut pas apparaître dans les spécifications d'héritage multiple. Si l'on ne précise pas la visibilité, elle est `public` par défaut si l'on définit une structure (`struct`), et `private` par défaut si l'on définit une classe (`class`). La déclaration ci-dessus est donc équivalente à:

```
class femme_mariee: femme, marie_ou_mariee { /* … */ };
```

Les constructeurs pour la nouvelle classe `femme_mariee` font appel aux **constructeurs des superclasses** en les *nommant explicitement*, comme dans:

```
class femme_mariee: public femme, public marie_ou_mariee
     {
public:
     //   d'autre déclarations…
                    femme_mariee (
                         int naissance, int mariage, int enfants )
                         // un constructeur

                         :    // constructeurs des deux superclasses
                         femme (naissance, enfants),
                         marie_ou_mariee (mariage)

                         { /* Ce corps est vide */ }
```

On ne peut s'appuyer sur les seuls profils des constructeurs comme '`(naissance, enfants)`' pour déterminer à quelle superclasse ils appartiennent, car il se peut qu'un profil donné corresponde à plusieurs constructeurs de superclasses différentes. De plus, l'ordre des spécifications d'héritage multiple influe sur l'ordre dans lequel les constructeurs et destructeurs seront déclenchés, comme on le verra au Aspects avancés du langage.

☞ D'une manière plus générale, **l'héritage multiple impose de bien spécifier certains points de sémantique**.

Par exemple, **si une fonction appartient à plusieurs super-classes, comment lever l'ambiguïté?** Ceci peut se faire en C++ à l'aide de l'opérateur d'accès à un niveau de déclaration ' : :', et:

```
humain :: ecrire ()
```

désigne la version de `ecrire` propre à humain dans touts ses descendants, sans ambigüité.

Un autre problème est que l'**on peut hériter plusieurs fois la même classe, même indirectement**, comme c'est le cas de `homme_veuf` dans notre exemple, qui hérite du type `humain` en tant qu'`homme` et en tant que `veuf_ou_veuve`.

Dans le cas de l'exemple ci-dessus, on évite cette '**duplication des membres des superclasses**' en déclararant que toutes les classes dans la hiérarchie sauf celles qui en constituent les feuilles - celles qui n'ont pas de descendance - héritent de `humain` en un seul exemplaire, avec la spécification `virtual`, comme dans:

```
class veuf_ou_veuve: virtual public humain { /* … */ }
```

On voit d'ailleurs là un **coté critiquable de l'économie mal-venue de mots clés**, donnant a `virtual` des sens assez différents selon l'emploi syntaxique qui en est fait.

☞ **Toutes les sous-classes `virtual` d'une classe `une_classe` partagent une même représentation en mémoire en tant que `une_classe`.**

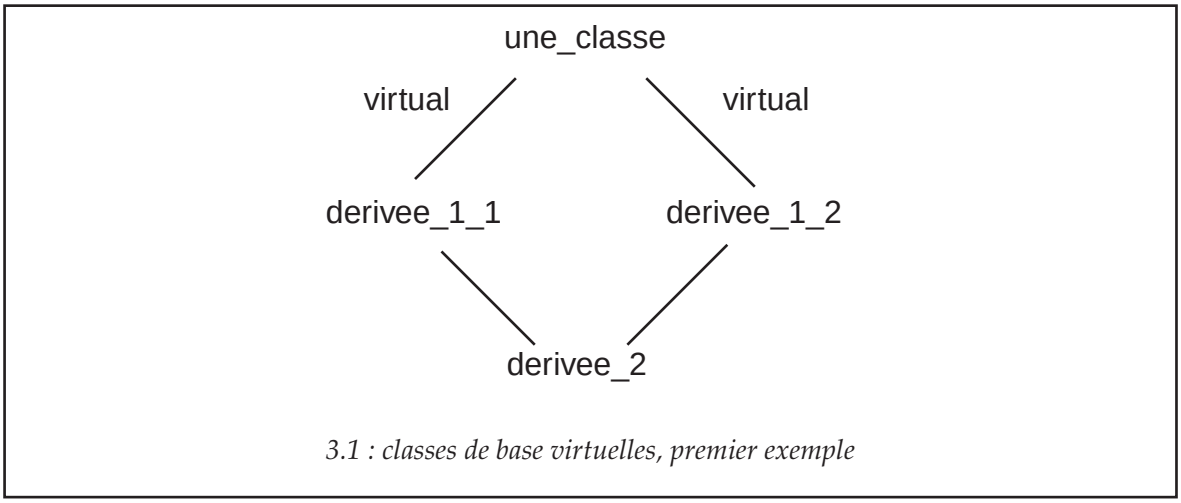
Ainsi dans l'exemple:

```
class derivee_1_1: virtual public une_classe { /* … */ };
class derivee_1_2: virtual public une_classe { /* … */ };

class derivee_2: public derivee_1_1, public derivee_1_2 { /* … */ };
```

décrit par le diagramme de la figure 3.1 , les instances de la classe `derivee_2` ne contiennent **qu'un sous-objet** de la classe `une_classe`.

En revanche, dans:

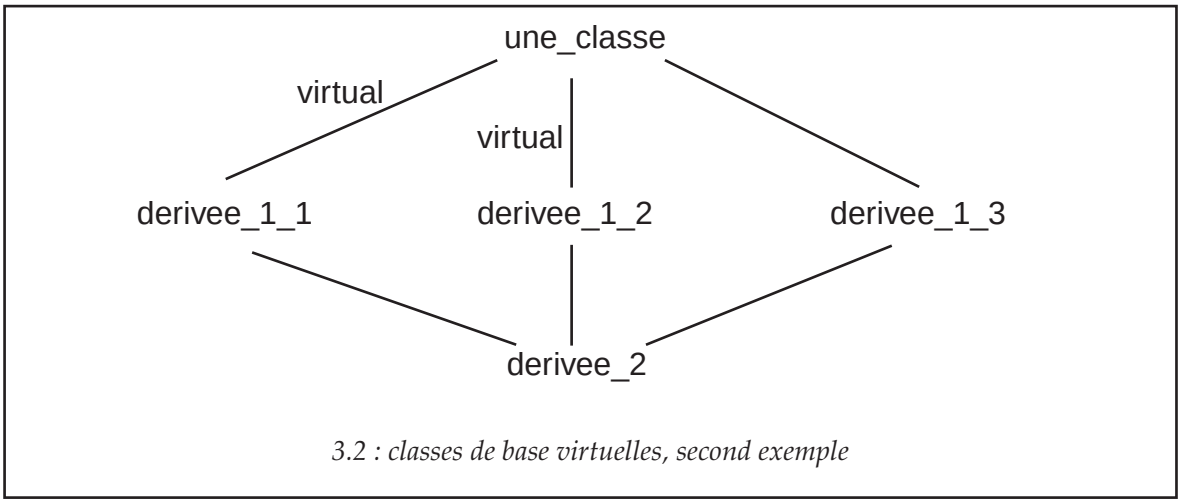
```
class derivee_1_1: virtual public une_classe { /* … */ };
class derivee_1_2: virtual public une_classe { /* … */ };
class derivee_1_3: public une_classe { /* … */ };

class derivee_2:
     public derivee_1_1, public derivee_1_2, public derivee_1_3
```



3.1 : classes de base virtuelles, premier exemple

```
{ // … };
```

illustré par la figure 3.1 , toute instance la classe `derivee_2` contient **deux sous-objets**



3.2 : classes de base virtuelles, second exemple

de la classe `une_classe`: l'un est celui de `derivee_1_3`, tandis que l'autre est celui partagé par `derivee_1_1` et `derivee_1_2`.

☞ On peut bien sûr lever l'ambiguïté sur l'accès aux membres et fonctions membres répétés à l'aide de l'opérateur '`::`' d'accès à un niveau de déclaration.

Un **exemple concret d'héritage multiple** est fourni dans le fichier **`stream.h`** faisant partie des libraries de C++, selon le schéma suivant:

```
class streambuf;                                    //   déclaration
class ostream;                                      //   déclaration

class ios { /* … */ };

class streambuf { // … };                            //   définition
```

```
class istream : virtual public ios { /* … */ };
class ostream : virtual public ios { /* … */ };     //   définition

class iostream : public istream, public ostream {
     public:
                                iostream (streambuf *);
          virtual               ~ iostream ();

     protected:
                                iostream ();
     };

class istream_withassign : public istream {
     public:
                                istream_withassign ();

          virtual               ~ istream_withassign ();

          istream_withassign    & operator = (istream &);
          istream_withassign    & operator = (streambuf *);
     };


class ostream_withassign : public ostream {
     public:
                                ostream_withassign ();

          virtual               ~ ostream_withassign ();

          ostream_withassign    & operator = (ostream &);
          ostream_withassign    & operator = (streambuf *);
     };


class iostream_withassign : public iostream {
     public:
                                iostream_withassign ();

          virtual               ~ iostream_withassign ();

          iostream_withassign   & operator = (ios &);
          iostream_withassign   & operator = (streambuf *);
     };


extern istream_withassign   cin;
extern ostream_withassign   cout;
extern ostream_withassign   cerr;
extern ostream_withassign   cdebug;
```

On voit dans cet exemple la **surcharge sémantique de l'opérateur** '=' comme dans:

```
istream_withassign      & operator = (istream &);
```

où l'on indique que l'on peut écrire:

```
istream_withassign      un_istream_withassign;
istream                 un_istream, un_autre_istream;
```

```
un_istream_withassign = un_istream;
```

et que le résultat de l'affectation est lui-même une valeur du type `istream_withassign &`, permettant d'écrire une **affectation multiple** comme:

```
un_istream_withassign = un_istream = un_autre_istream;
```

de la même manière que l'on peut cascader les emplois de << et >> pour les entrées/sorties sur des flots.

La surcharge des opérateurs sera traitée plus en détail dans le Aspects avancés du langage.

UNIVERSITÉ DE GENÈVE

Table des matières

Index

Symboles

Numériques

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X