Université Claude Bernard – Lyon 1 Premier Cycle Sciences

O. Marguin

C++ et programmation-objet

– COURS –

U.E. 33 — option 6 1999 – 2000

INTRODUCTION

Apparue au début des années 70, la programmation orientée objet répond aux nécessités de l'informatique professionnelle. Elle offre aux concepteurs de logiciels une grande souplesse de travail, permet une maintenance et une évolution plus aisée des produits.

Mais sa pratique passe par une approche radicalement différente des méthodes de programmation traditionnelles : avec les langages à objets, le programmeur devient metteur en scène d'un jeu collectif où chaque objet-acteur se voit attribuer un rôle bien précis.

Le présent document explique les règles de ce jeu. Ces règles sont mises en pratique en C++, ce qui permet une initiation naturelle à ce langage très utilisé de nos jours. Les exemples sont donnés pour l'environnement de programmation Turbo C++ *de Borland.*

TABLE DES MATIERES

Chapitre 1 : Le concept d'objet 1

1.1 Objet usuel 1
1.2 Objet informatique – Classe 3
1.3 Encapsulation 4
1.4 Stratégie D.D.U 4
1.5 Exemple 6

Chapitre 2 : Les ingrédients des classes 9

2.1 Variables 9
2.2 Expressions 10
2.3 Instructions 12
2.4 Fonctions 13
2.5 Exemple 15
2.6 Instructions conditionnelles 17
2.7 Boucles 19
2.8 Compléments 21

Chapitre 3 : Programmation des classes 23

3.1 Fonctions-membres 23
3.2 Constructeurs et destructeurs 24
3.3 Surcharge des opérateurs 26
3.4 Réalisation d'un programme 28

Chapitre 4 : Les ingrédients des classes (suite) 33

4.1 Enumérations 33
4.2 Tableaux 33
4.3 Chaînes de caractères 35
4.4 Pointeurs 37
4.5 Récapitulatif des opérateurs 42

Chapitre 5 : Héritage 43

5.1 Relations *a-un, est-un, utilise-un* 43
5.2 Classes dérivées 43
5.3 Polymorphisme 47
5.4 Exemple 49
5.5 Héritage multiple 55
5.6 Classes abstraites 56

Chapitre 6 : Entrées & sorties 59

6.1 La librairie `iostream.h` 59
6.2 La librairire `fstream.h` 62
6.3 Fonctions de contrôle 63
6.4 Surcharge des opérateurs `>>` et `<<` 64
6.5 Formatage des données 65

Appendice 67

A.1 Relation d'amitié 67
A.2 Patrons 68

Chapitre 1

LE CONCEPT D'OBJET

1.1 Objet usuel

(1.1.1) Comment décrire un objet usuel ? Prenons exemple sur la notice d'utilisation d'un appareil ménager. Cette notice a généralement trois parties :

a. une description physique de l'appareil et de ses principaux éléments (boutons, voyants lumineux, cadrans etc.), schémas à l'appui,

b. une description des fonctions de chaque élément,

c. un mode d'emploi décrivant la succession des manœuvres à faire pour utiliser l'appareil.

Seules les parties a et b sont intrinsèques à l'appareil : la partie c concerne l'utilisateur et rien n'empêche celui-ci de se servir de l'appareil d'une autre manière, ou à d'autres fins que celles prévues par le constructeur.

Nous retiendrons donc que pour décrire un objet usuel, il faut décrire ses composants, à savoir :

1. *les différents éléments qui le constituent,*
2. *les différentes fonctions associées à ces éléments.*

(1.1.2) Les éléments qui constituent l'objet définissent à chaque instant l'*état* de l'objet — on peut dire : son aspect spatial. Les fonctions, quant à elles, définissent le *comportement* de l'objet au cours du temps. Il serait intéressant d'approfondir cette dualité espace-temps à propos des objets, mais cela nous entraînerait sur un terrain philosophique hors de propos ici.

Les éléments qui constituent l'objet peuvent se modifier au cours du temps (par exemple, le voyant d'une cafetière peut être allumé ou éteint). Un objet peut ainsi avoir plusieurs états. Le nombre d'états possibles d'un objet donne une idée de sa *complexité*.

(1.1.3) Pour identifier les composants d'un objet usuel, une bonne méthode consiste à faire de cet objet une description littérale (*méthode Proust*), puis de souligner les principaux noms communs et verbes. Les noms communs donnent les éléments constituants, les verbes donnent les fonctions.

Illustrons cette méthode dans le cas d'un objet très simple, un marteau :

On peut en faire la description suivante :

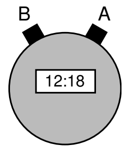
> "Ce marteau comporte un manche en bois, une extrêmité plate en métal et une extrêmité incurvée également en métal. Le manche permet de saisir le marteau, l'extrêmité plate permet de frapper quelque chose et l'extrêmité incurvée permet d'arracher quelque chose."

D'où la *fiche descriptive* :

nom : marteau	
éléments :	*fonctions :*
manche	saisir
extrêmité plate	frapper
extrêmité incurvée	arracher

Pour vérifier que nous n'avons rien oublié d'important dans une telle fiche descriptive, il faut imaginer l'objet à l'œuvre dans une petite scène (*méthode Feydeau*). Le déroulement de l'action peut alors révéler des composants qui nous auraient échappé en première analyse (dans notre exemple, quatre acteurs : un marteau, un clou, un mur et un individu ; pour planter le clou dans le mur, l'individu saisit le marteau par son manche, puis frappe sur le clou avec l'extrêmité plate ; il s'aperçoit alors que le clou est mal placé, et l'arrache avec l'extrêmité incurvée).

(1.1.4) Prenons comme deuxième exemple un chronomètre digital :



> "Ce chronomètre comporte un temps qui s'affiche et deux boutons A et B. Quand on presse sur A, on déclenche le chronomètre, ou bien on l'arrête. Quand on presse sur B, on remet à zéro le chronomètre."

D'où la *fiche descriptive* :

nom : chronomètre	
éléments :	*fonctions :*
boutons A, B	afficher
temps	presser sur un bouton
	déclencher
	arrêter
	remettre à zéro

Remarquons que l'utilisateur ne peut pas modifier directement le temps affiché : il n'a accès à ce temps que de manière indirecte, par l'intermédiaire des fonctions de l'objet. Cette notion d'accès indirect jouera un rôle important dans la suite (1.3).

1.2 Objet informatique — Classe

(1.2.1) L'ordinateur est un appareil possédant une très grande complexité — liée à un très grand nombre d'états — et un comportement très varié — lié à la façon dont on le programme. Il nous servira d'*objet universel* capable de simuler la plupart des objets usuels.

(1.2.2) Programmer un ordinateur, c'est lui fournir une série d'instructions qu'il doit exécuter. Ces instructions sont généralement écrites dans un langage dit *évolué*, plus maniable que le langage machine, puis traduites automatiquement en langage machine (cette traduction s'appelle *compilation*).

Un langage de programmation évolué doit simplifier le travail du programmeur en lui offrant la possibilité :

- d'écrire son programme sous forme de petits modules autonomes,
- de corriger et faire évoluer son programme avec un minimum de retouches,
- d'utiliser des modules tout faits et fiables.

De ce point de vue, les langages à objets comme le C++ sont supérieurs aux langages classiques comme le C, car ils font reposer le gros du travail sur des "briques logicielles intelligentes" : les objets. Un programme n'est alors qu'une collection d'objets mis ensemble par le programmeur et qui coopèrent, exactement comme les joueurs d'une équipe de football supervisés par leur entraîneur.

(1.2.3) Transposé en langage informatique, (1.1.1) donne :

Un objet est une structure informatique regroupant :

- *des variables, caractérisant l'état de l'objet,*
- *des fonctions, caractérisant le comportement de l'objet.*

Les variables (resp. fonctions) s'appellent *données-membres* (resp. *fonctions-membres*) de l'objet. L'originalité dans la notion d'objet, c'est que variables et fonctions sont regroupées dans une même structure.

(1.2.4) *Un ensemble d'objets de même type s'appelle une classe.*

Tout objet appartient à une classe, on dit aussi qu'il est une *instance* de cette classe. Par exemple, si l'on dispose de plusieurs chronomètres analogues à celui décrit en (1.1.4), ces chronomètres appartiennent tous à une même classe "chronomètre", chacun est une instance de cette classe. En décrivant la classe "chronomètre", on décrit la structure commune à tous les objets appartenant à cette classe.

(1.2.5) *Pour utiliser les objets, il faut d'abord décrire les classes auxquelles ces objets appartiennent.*

La description d'une classe comporte deux parties :

- une partie *déclaration*, fiche descriptive des données et fonctions-membres des objets de cette classe, qui servira d'interface avec le monde extérieur,
- une partie *implémentation*, contenant la programmation des fonctions-membres.

1.3 Encapsulation

(1.3.1) Dans la déclaration d'une classe, il est possible de protéger certaines données-membres ou fonctions-membres en les rendant invisibles de l'extérieur : c'est ce qu'on appelle l'*encapsulation*.

A quoi cela sert-il ? Supposons qu'on veuille programmer une classe `Cercle` avec comme données-membres :

- un point représentant le `centre`,
- un nombre représentant le `rayon`,
- un nombre représentant la `surface` du cercle.

Permettre l'accès direct à la variable `surface`, c'est s'exposer à ce qu'elle soit modifiée depuis l'extérieur, et cela serait catastrophique puisque l'objet risquerait alors de perdre sa cohérence (la surface dépend en fait du rayon). Il est donc indispensable d'interdire cet accès, ou au moins permettre à l'objet de le contrôler.

(1.3.2) *Données et fonctions-membres d'un objet O seront déclarées publiques si on autorise leur utilisation en dehors de l'objet O, privées si seul l'objet O peut y faire référence.*

Dans la déclaration d'une classe, comment décider de ce qui sera public ou privé ? Une approche simple et sûre consiste à déclarer systématiquement les données-membres *privées* et les fonctions-membres *publiques*. On peut alors autoriser l'accès aux données-membres (pour consultation ou modification) par des fonctions prévues à cet effet, appelées *fonctions d'accès*.

Ainsi, la déclaration de la classe `Cercle` ci-dessus pourrait ressembler à :

classe : `Cercle`	
privé :	*public :*
`centre`	`Fixer_centre`
`rayon`	`Fixer_rayon`
`surface`	`Donner_surface`
	`Tracer`

Dans le cas d'une classe `chronometre` (1.1.4), il suffirait de ne déclarer publiques que les seules fonctions-membres `Afficher` et `Presser_sur_un_bouton` pour que les chronomètres puissent être utilisés normalement, en toute sécurité.

1.4 Stratégie D.D.U

(1.4.1) *En C++, la programmation d'une classe se fait en trois phases : déclaration, définition, utilisation (en abrégé : D.D.U).*

Déclaration : c'est la partie interface de la classe. Elle se fait dans un fichier dont le nom se termine par `.H` Ce fichier se présente de la façon suivante :

```
class Maclasse
{
public:
        déclarations des données et fonctions-membres publiques

private:
        déclarations des données et fonctions-membres privées
};
```

Définition : c'est la partie implémentation de la classe. Elle se fait dans un fichier dont le nom se termine par `.CPP` Ce fichier contient les définitions des fonctions-membres de la classe, c'est-à-dire le code complet de chaque fonction.

Utilisation : elle se fait dans un fichier dont le nom se termine par `.CPP`

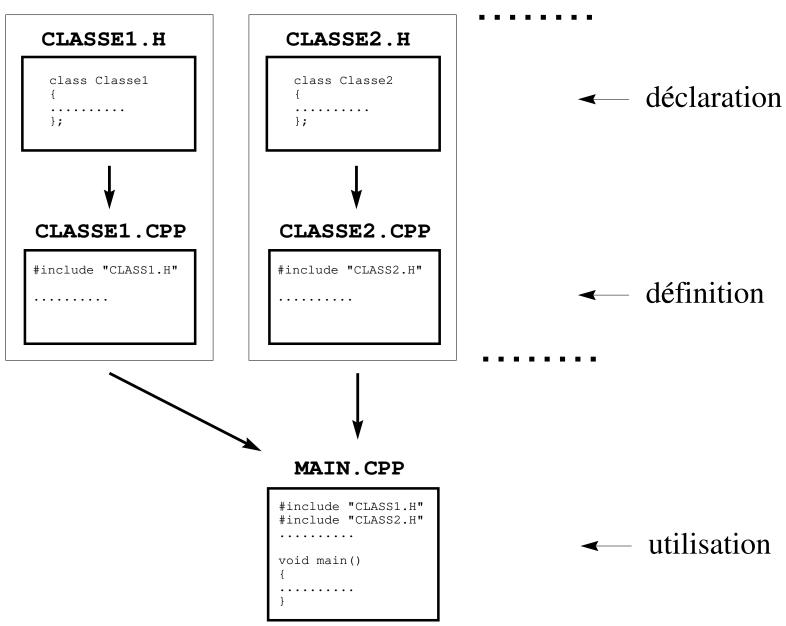
(1.4.2) Structure d'un programme en C++

Nos programmes seront généralement composés d'un nombre impair de fichiers :

- pour chaque classe :
  - un fichier `.H` contenant sa déclaration,
  - un fichier `.CPP` contenant sa définition,
- un fichier `.CPP` contenant le traitement principal.

Ce dernier fichier contient une fonction appelée `main` : c'est par cette fonction que commence l'exécution du programme.

Schématiquement :



(1.4.3) La *directive d'inclusion* `#include` permet d'inclure un fichier de déclarations dans un autre fichier : si le fichier à inclure s'appelle `untel.h`, on écrira `#include`

<untel.h> s'il s'agit d'un fichier standard livré avec le compilateur C++, ou #include "untel.h" s'il s'agit d'un fichier écrit par nous-mêmes.

1.5 Exemple

Bien que cela anticipe sur les éléments de syntaxe fournis au chapitre 2, nous donnons ici un programme complet afin d'illustrer les principes exposés au paragraphe précédent. Ce programme simule le fonctionnement d'un parcmètre.

(1.5.1) Le programme se compose de trois fichiers :

> PARCMETR.H qui contient la déclaration de la classe Parcmetre,
> PARCMETR.CPP qui contient la définition de la classe Parcmetre,
> SIMUL.CPP qui contient l'utilisation de la classe Parcmetre.

```
// ----------------------- parcmetr.h -----------------------
// ce fichier contient la déclaration de la classe Parcmetre

class Parcmetre
{
public:
    Parcmetre();                        // constructeur de la classe
    void Affiche();                     // affichage du temps de stationnement
    void PrendsPiece(int valeur);       // introduction d'une pièce
private:
    int heures,                         // chiffre des heures...
        minutes;                        // et des minutes
};
```

```
// ----------------------- parcmetr.cpp ---------------------
// ce fichier contient la définition de la classe Parcmetre

#include <iostream.h>           // pour les entrées-sorties
#include "parcmetr.h"           // déclaration de la classe Parcmetre

Parcmetre::Parcmetre()          // initialisation d'un nouveau parcmètre
{
    heures = minutes = 0;
}

void Parcmetre::Affiche()       // affichage du temps de stationnement restant
                                // et du mode d'emploi du parcmètre
{
    cout << "\n\n\tTEMPS DE STATIONNEMENT :";
    cout << heures << " heures  " << minutes << " minutes";
    cout << "\n\n\nMode d'emploi du parcmètre :";
    cout << "\n\tPour mettre une pièce de 1 F  : tapez A";
    cout << "\n\tPour mettre une pièce de 2 F  : tapez B";
    cout << "\n\tPour mettre une pièce de 5 F  : tapez C";
    cout << "\n\tPour mettre une pièce de 10 F : tapez D";
    cout << "\n\tPour quitter le programme     : tapez Q";
}
```

```
void Parcmetre::PrendsPiece(int valeur)       // introduction d'une pièce
{
    minutes += valeur * 10;            // un franc = 10 minutes de stationnement
    while (minutes >= 60)
    {
        heures += 1;
        minutes -= 60;
    }
    if (heures >= 3)           // on ne peut dépasser 3 heures
    {
        heures = 3;
        minutes = 0;
    }
}



// ----------------------- simul.cpp --------------------
// ce fichier contient l'utilisation de la classe Parcmetre

#include <iostream.h>          // pour les entrées-sorties
#include "parcmetr.h"          // pour la déclaration de la classe Parcmetre

void main()                    // traitement principal
{
    Parcmetre p;               // déclaration d'un parcmètre p
    char choix = 'X';

    while (choix != 'Q')       // boucle principale d'événements
    {
        p.Affiche();
        cout << "\nchoix ? --> ";
        cin >> choix;          // lecture d'une lettre
        switch (choix)         // action correspondante
        {
            case 'A' :
                p.PrendsPiece(1);
                break;
            case 'B' :
                p.PrendsPiece(2);
                break;
            case 'C' :
                p.PrendsPiece(5);
                break;
            case 'D' :
                p.PrendsPiece(10);
        }
    }
}
```

(1.5.2) **Remarques**

Commentaires : pour rendre un programme plus lisible, il est conseillé d'y mettre des commentaires. En C++, tout texte qui suit // jusqu'à la fin de la ligne est un commentaire.

Conventions typographiques : contrairement au Pascal, le C++ distingue les minuscules et les majuscules. L'usage veut que l'ensemble d'un programme soit écrit en minuscules. Cependant les noms des classes ainsi que des fonctions-membres des classes commencent généralement par une majuscule.

(1.5.3) **Réalisation pratique du programme**

Elle se fait en trois étapes :

1) création des fichiers sources `PARCMETR.H`, `PARCMETR.CPP` et `SIMUL.CPP`.

2) compilation des fichiers `.CPP`, à savoir `PARCMETR.CPP` et `SIMUL.CPP`, ce qui crée deux fichiers objets `PARCMETR.OBJ` et `SIMUL.OBJ` (ces fichiers sont la traduction en langage machine des fichiers `.CPP` correspondants),

3) édition des liens entre les fichiers objets, pour produire finalement un fichier exécutable dont le nom se termine par `.EXE`. Cet exécutable est indépendant de l'environnement de programmation et peut être lancé directement depuis le système d'exploitation (DOS).

Dans l'environnement Turbo C++ de Borland, les phases 2 et 3 sont automatisées grâce à la gestion d'un *projet* : il suffit de créer un projet qu'on appelle par exemple `PARCM.PRJ`, d'y inclure les fichiers `.CPP` et de demander ensuite la construction du projet (commande `Make`). L'exécutable final se nomme alors `PARCM.EXE`.

Remarque.— On peut inclure directement dans un projet un fichier `.OBJ` : il n'est pas nécessaire de disposer du fichier source `.CPP` correspondant. On pourra donc travailler avec des classes déjà compilées.

Chapitre 2

LES INGREDIENTS DES CLASSES

2.1 Variables

(2.1.1) Terminologie

Une *variable* est caractérisée par :
- son *nom* : mot composé de lettres ou chiffres, commençant par une lettre (le caractère _ tient lieu de lettre),
- son *type* précisant la nature de cette variable (nombre entier, caractère, objet etc.),
- sa *valeur* qui peut être modifiée à tout instant.

Durant l'exécution d'un programme, à toute variable est attachée une *adresse* : nombre entier qui indique où se trouve stockée en mémoire la valeur de cette variable.

(2.1.2) Types de base

a) **vide** : `void` . Aucune variable ne peut être de ce type. On en verra l'usage au paragraphe (2.4.2).

b) **entiers**, par taille-mémoire croissante :
   - `char`, stocké sur un octet ; valeurs : de -2^7 à $2^7 - 1$ (-128 à 127),
   - `short`, stocké sur 2 octets ; valeurs : de -2^{15} à $2^{15} - 1$ (-32768 à 32767),
   - `long`, stocké sur 4 octets ; valeurs : de -2^{31} à $2^{31} - 1$,
   - `int`, coïncide avec `short` ou `long`, selon l'installation.

c) **réels**, par taille-mémoire croissante :
   - `float`, stocké sur 4 octets ; précision : environ 7 chiffres,
   - `double`, stocké sur 8 octets ; précision : environ 15 chiffres,
   - `long double`, stocké sur 10 octets ; précision : environ 18 chiffres.

(2.1.3) Valeurs littérales (ou explicites)

a) caractères usuels entre apostrophes, correspondant à des entiers de type `char`. Par exemple : `'A'` (= 65), `'a'` (= 97), `'0'` (= 48), `' '` (= 32). La correspondance caractères usuels $\leftrightarrow$ entiers de type `char` est donnée par la *table des codes ASCII*.
A connaître les caractères spéciaux suivants :
   `'\n'` : retour à la ligne,
   `'\t'` : tabulation.

b) chaînes de caractères entre guillemets, pour les affichages.

Par exemple : `"Au revoir!\n"`.
On reviendra plus tard sur le type *chaîne de caractères.*

c) valeurs entières, par exemple : `123` , `-25000` , `133000000` (ici respectivement de type `char`, `short`, `long`).

d) valeurs réelles, qui comportent une virgule, par exemple : `3.14`, `-1.0`, `4.21E2` (signifiant 4,21 10^2).

(2.1.4) **Déclaration des variables**

En C++, toute variable doit être déclarée avant d'être utilisée.

Forme générale d'une déclaration : <*type*> <*liste de variables*>`;`

où :
<*type*> est un nom de type ou de classe,
<*liste de variables*> est un ou plusieurs noms de variables, séparés par des virgules.

Exemples :

```
int i, j, k;     // déclare trois entiers i, j, k
float x, y;      // déclare deux réels x, y
Parcmetre p;     // déclare un parcmètre p, instance de la classe Parcmetre
```

(2.1.5) **Déclaration + initialisation**

En même temps qu'on déclare une variable, il est possible de lui attribuer une valeur initiale. On pourra écrire par exemple :

```
int i, j = 0, k;
float x, y = 1.0;
```

En particulier, on peut déclarer une *constante* en ajoutant `const` devant le nom du type, par exemple :

```
const double PI = 3.14159265358979323846;
const int MAX = 100;
```

Les valeurs des constantes ne peuvent pas être modifiées.

(2.1.6) **Cas des données-membres d'une classe**

Supposons que `mon_objet` soit une instance de `Ma_classe`. Pour désigner les données-membres de `mon_objet`, on utilise une *notation pointée* : on écrit `mon_objet.donnee` (à condition que `donnee` figure dans la déclaration de `Ma_classe`, et que l'accès en soit possible : voir (1.3)).

2.2 Expressions

(2.2.1) **Définition**

En combinant des noms de variables, des opérateurs, des parenthèses et des appels de fonctions (voir (2.4)), on obtient des *expressions.*

Une règle simple à retenir :

En C++, on appelle expression tout ce qui a une valeur.

(2.2.2) Opérateurs arithmétiques

- `+` : addition,
- `-` : soustraction,
- `*` : multiplication,
- `/` : division. Attention : entre deux entiers, donne le quotient entier,
- `%` : entre deux entiers, donne le reste modulo.

Exemples :

`19.0 / 5.0`	vaut 3.8,
`19 / 5`	vaut 3,
`19 % 5`	vaut 4.

Dans les expressions, les *règles de priorité* sont les règles usuelles des mathématiciens. Par exemple, l'expression `5 + 3 * 2` a pour valeur `11` et non `16`. En cas de doute, il ne faut pas hésiter à mettre des parenthèses.

`++` : incrémentation. Si `i` est de type entier, les expressions `i++` et `++i` ont toutes deux pour valeur la valeur de `i`. Mais elles ont également un *effet de bord* qui est :
- pour la première, d'ajouter ensuite 1 à la valeur de `i` (*post-incrémentation*),
- pour la seconde, d'ajouter d'abord 1 à la valeur de `i` (*pré-incrémentation*).

`--` : décrémentation. `i--` et `--i` fonctionnent comme `i++` et `++i`, mais retranchent 1 à la valeur de `i` au lieu d'ajouter 1.

(2.2.3) Opérateurs d'affectation

`=` (égal)

Forme générale de l'expression d'affectation : <*variable*> `=` <*expression*>

Fonctionnement :

1. l'<*expression*> est d'abord évaluée ; cette valeur donne la valeur de l'expression d'affectation,
2. effet de bord : la <*variable*> reçoit ensuite cette valeur.

Exemples :

`i = 1`
`i = j = k = 1` (vaut 1 et donne à `i`, `j` et `k` la valeur 1)
`mon_obj.donnee = 0` (si `donnee` est accessible).

`+=`, `-=`, `*=`, `/=`, `%=`

Forme générale : <*variable*> <*opérateur*>`=` <*expression*>

L'expression est équivalente à :
<*variable*> `=` <*variable*> <*opérateur*> <*expression*>

Par exemple, l'expression `i += 3` équivaut à `i = i + 3`.

Remarque : l'expression d'affectation peut provoquer une conversion de type ! Par exemple, supposons déclarés :

```
int i;
float x;
```

Alors, si `i` vaut 3, l'expression `x = i` donne à `x` la valeur 3.0 (conversion entier → réel). Inversement, si `x` vaut 4.21, l'expression `i = x` donne à `i` la valeur 4, partie entière de `x` (conversion réel → entier).

(2.2.4) **Opérateurs d'entrées-sorties**

Ce sont les opérateurs `<<` et `>>`, utilisés en conjonction avec les objets prédéfinis `cout` et `cin` déclarés dans `<iostream.h>` (ne pas oublier la directive `#include <iostream.h>` en début de fichier).

Formes :

`cout <<` <*expression*> : affichage à l'écran de la valeur de <*expression*>,
`cin >>` <*variable*> : lecture au clavier de la valeur de <*variable*>

2.3 Instructions

(2.3.1) **Instruction-expression**

Forme : <*expression*>;

Cette instruction n'est utile que si l'<*expression*> a un effet de bord.

Exemples :

```
i++;
5;              // correct, mais sans intérêt !
;               // instruction vide
i += j = 3;     // ce genre d'instruction est à éviter !
```

(2.3.2) **Instruction-bloc**

Forme : {
<*déclarations et instructions*>
}

Exemple :

```
{
        int i = 3, j;
        double x = 2.2, y = 3.3;
        y = 0.5 * (x + y);
        int k = 1 - (j = i);
}
```

(2.3.3) **Structures de contrôle**

Ce sont des instructions qui permettent de contrôler le déroulement des opérations effectuées par l'ordinateur : instructions `if` et `switch` (branchements conditionnels), instructions `while`, `do` et `for` (boucles). Se reporter aux paragraphes (2.6) et (2.7).

(2.3.4) **Visibilité d'une variable**

Le domaine de visibilité d'une variable est limité au bloc où cette variable est déclarée, et après sa déclaration.

On dit aussi que les variables déclarées dans un bloc sont *locales* à ce bloc.

Si une variable est locale à un bloc B, on ne peut y faire référence en dehors de ce bloc, mais on peut l'utiliser à l'intérieur de B et dans tout bloc inclus lui-même dans B.

Dans un même bloc, il est interdit de déclarer deux variables avec le même nom. Mais cela est possible dans deux blocs distincts, même si l'un des blocs est inclus dans l'autre.

Lorsque dans un fichier, une variable est déclarée en dehors de tout bloc (c'est-à-dire au *niveau principal*), on dit qu'elle est *globale* ; elle est alors visible de tout le fichier, à partir de l'endroit où elle est déclarée. Cette règle de visibilité s'appliquera également aux fonctions.

2.4 Fonctions

(2.4.1) **Terminologie**

En C++, la partie exécutable d'un programme (c'est-à-dire celle qui comporte des instructions) n'est composée que de fonctions. Chacune de ces fonctions est destinée à effectuer une tâche précise et renvoie généralement une valeur, résultat d'un calcul.

Une *fonction* est caractérisée par :

1) son *nom*,
2) le *type* de valeur qu'elle renvoie,
3) l'information qu'elle reçoit pour faire son travail (*paramètres*),
4) l'instruction-bloc qui effectue le travail (*corps* de la fonction).

Les trois premiers éléments sont décrits dans la *déclaration* de la fonction. L'élément n°4 figure dans la *définition* de la fonction.

Toute fonction doit être déclarée et définie avant d'être utilisée.

Une fonction est

- soit *membre* d'une classe (sa déclaration fait partie de la déclaration de la classe)
- soit *libre* (sa déclaration figure alors au niveau principal)

La définition d'une fonction se fait toujours au niveau principal. On ne peut donc pas imbriquer les fonctions les unes dans les autres, comme on le fait en Pascal.

(2.4.2) **Déclaration d'une fonction libre**

Elle se fait grâce à un *prototype* de la forme suivante :

<*type*> <*nom*>`(`<*liste de paramètres formels*>`)` ;

où <*type*> est le type du résultat, <*nom*> est le nom de la fonction et <*liste de paramètres formels*> est composé de zéro, une ou plusieurs déclarations de variables, séparées par des virgules.

Exemples :

```
double Moyenne(double x, double y);
char LireCaractere();
void AfficherValeurs(int nombre, double valeur);
```

Remarque : la dernière fonction n'est pas destinée à renvoyer une valeur ; c'est pourquoi le type du résultat est `void` (une telle fonction est parfois appelée *procédure*).

(2.4.3) **Définition d'une fonction libre**

Elle est de la forme suivante :

> <*type*> <*nom*>(<*liste de paramètres formels*>)
> <*instruction-bloc*>

Donnons par exemple les définitions des trois fonctions déclarées ci-dessus :

```
double Moyenne(double x, double y)
{
        return (x + y) / 2.0;
}

char LireCaractere()
{
        char c;
        cin >> c;
        return c;
}

void AfficherValeurs(int nombre, double valeur)
{
        cout << '\t' << nombre << '\t' << valeur << '\n';
}
```

A retenir :

L'instruction **`return`** <*expression*>; *interrompt l'exécution de la fonction. La valeur de l'*<*expression*> *est la valeur que renvoie la fonction.*

(2.4.4) **Utilisation d'une fonction libre**

Elle se fait grâce à l'*appel* de la fonction. Cet appel est une expression de la forme : <*nom*>(<*liste d'expressions*>).

Mécanisme de l'appel :

- chaque expression de la <*liste d'expressions*> est évaluée,
- les valeurs ainsi obtenues sont transmises dans l'ordre aux paramètres formels,
- le corps de la fonction est ensuite exécuté,
- la valeur renvoyée par la fonction donne le résultat de l'appel.

Si la fonction ne renvoie pas de valeur, le résultat de l'appel est de type `void`.

Voici un bout de programme avec appel des trois fonctions précédentes :

```
double u, v;
cout << "\nEntrez les valeurs de u et v :";
cin >> u >> v;
double m = Moyenne(u, v);
cout << "\nVoulez-vous afficher la moyenne ?  ";
char reponse = LireCaractere();
if (reponse == 'o')
        AfficherValeurs(2, m);
```

(2.4.5) **Arguments par défaut**

On peut, lors de la déclaration d'une fonction, choisir pour les paramètres des valeurs par défaut (sous forme de déclarations-initialisations figurant à la fin de la liste des paramètres formels). Par exemple :

```
void AfficherValeurs(int nombre, double valeur = 0.0);
```

Les deux appels suivants sont alors corrects :

```
AfficherValeurs(n, x);
AfficherValeurs(n);  // équivaut à : AfficherValeurs(n, 0.0);
```

(2.4.6) **Cas des fonctions-membres**

Ce qui vient d'être dit s'applique aussi aux fonctions-membres d'une classe, à ceci près que :

- dans la définition de la fonction, on doit ajouter <*nom de la classe*>`::` devant le nom de la fonction :
  <*type*> <*nom de la classe*>`::`<*nom*>`(`<*liste de paramètres*>`)`
  <*instruction-bloc*>
- l'appel se fait avec la notation pointée (comparer avec (2.1.6) pour les données-membres), par exemple : `mon_obj.fonction()` ; en programmation-objet, on dit parfois qu'on envoie le *message* `fonction()` à l'objet *destinataire* `mon_obj`.

Exceptions : certaines fonctions-membres sont déclarées sans type de résultat et ont le même nom que celui de la classe : ce sont les constructeurs. Ces constructeurs permettent notamment d'initialiser les objets dès leur déclaration.

2.5 Exemple

(2.5.1) Pour illustrer ce qui précède, voici la déclaration et la définition d'une classe `Complexe` decrivant les nombres complexes, et un programme qui en montre l'utilisation.

```
// ----------------------- complexe.h -----------------------
//             déclaration de la classe Complexe

class Complexe
{
public:
    Complexe(float x, float y);       // premier constructeur de la classe :
                                      // fixe la partie réelle à x, la partie imaginaire à y
    Complexe();                       // second constructeur de la classe :
                                      // initialise un nombre complexe à 0
    void Lis();                       // lit un nombre complexe entré au clavier
    void Affiche();                   // affiche un nombre complexe
    Complexe operator+(Complexe g);   // surcharge de l'opérateur d'addition +
private:
    float re, im;                     // parties réelle et imaginaire
};
```

```
// ------------------------ complexe.cpp ---------------------
//               définition de la classe Complexe

#include <iostream.h>          // pour les entrées-sorties
#include "complexe.h"          // déclaration de la classe Complexe

Complexe::Complexe(float x, float y)   // constructeur avec paramètres
{
    re = x;
    im = y;
}

Complexe::Complexe()                   // constructeur sans paramètre
{
    re = 0.0;
    im = 0.0;
}

void Complexe::Lis()                   // lecture d'un complexe
{
    cout << "Partie réelle ?  ";
    cin >> re;
    cout << "Partie imaginaire ?  ";
    cin >> im;
}

void Complexe::Affiche()               // affichage d'un complexe
{
    cout << re << " + i " << im;
}

Complexe Complexe::operator+(Complexe g)       // surcharge de l'opérateur +
{
    return Complexe(re + g.re, im + g.im);     // appel du constructeur
}
```

```
// ------------------------ usage.cpp ---------------------
//         exemple d'utilisation de la classe Complexe

#include <iostream.h>          // pour les entrées-sorties
#include "complexe.h"          // pour la déclaration de la classe Complexe

void main()                    // traitement principal
{
    Complexe z1(0.0, 1.0);     // appel implicite du constructeur paramétré
    Complexe z2;               // appel implicite du constructeur non paramétré

    z1.Affiche();              // affichage de z1
    cout << "\nEntrer un nombre complexe : ";
    z2.Lis();                  // saisie de z2
    cout << "\nVous avez entré : ";
    z2.Affiche();              // affichage de z2

    Complexe z3 = z1 + z2;     // somme de deux complexes grâce à l'opérateur +
    cout << "\n\nLa somme de ";
    z1.Affiche();
    cout << " et ";
    z2.Affiche();
    cout << " est ";
    z3.Affiche();
}
```

(2.5.2) **Remarques**

Les constructeurs permettent d'initialiser les objets. Nous reviendrons sur leur usage au chapitre 3.

Nous reviendrons également sur la surcharge des opérateurs. Dans ce programme, nous donnons l'exemple de l'opérateur `+` qui est redéfini pour permettre d'additionner deux nombres complexes. Cela permet ensuite d'écrire tout simplement `z3 = z1 + z2` entre nombre complexes. Cette possibilité de redéfinir (on dit *surcharger*) les opérateurs usuels du langage est un des traits importants du C++.

2.6 Instructions conditionnelles

(2.6.1) **L'instruction if**

Forme : `if (`<*expression entière*>`)`
        <*instruction1*>
`else`
        <*instruction2*>

Mécanisme : l'<*expression entière*> est évaluée.
- si sa valeur est $\neq 0$, l'<*instruction1*> est effectuée,
- si sa valeur est nulle, l'<*instruction2*> est effectuée.

En C++, il n'y a pas de type *booléen* (c'est-à-dire logique). On retiendra que :

Toute expression entière $\neq 0$ *(resp. égale à* 0*) est considérée comme vraie (resp. fausse).*

Remarque 1.— La partie `else` <*instruction2*> est facultative.

Remarque 2.— <*instruction1*> et <*instruction2*> sont en général des instructions-blocs pour permettre d'effectuer plusieurs actions.

(2.6.2) **Opérateurs booléens**

`!` : non,
`&&` : et,
`||` : ou.

L'évaluation des expressions booléennes est une *évaluation courte*. Par exemple, l'expression `u && v` prend la valeur 0 dès que `u` est évalué à 0, sans que `v` ne soit évalué.

(2.6.3) **Opérateurs de comparaison**

`==` : égal,
`!=` : différent,
`<` : strictement inférieur,
`>` : strictement supérieur,
`<=` : inférieur ou égal,
`>=` : supérieur ou égal.

(2.6.4) **Exemple**

On veut définir une fonction calculant le maximum de trois entiers.

Première solution :

```
int Max(int x, int y, int z)   // calcule le maximum de trois entiers
{
        if ((x <= z) && (y <= z))
                return z;
        if ((x <= y) && (z <= y))
                return y;
        return x;
}
```

Deuxième solution, avec une fonction auxiliaire calculant le maximum de deux entiers :

```
int Max(int x, int y)          // calcule le maximum de deux entiers
{
        if (x < y)
                return y;
        return x;
}

int Max(int x, int y, int z)   // calcule le maximum de trois entiers
{
        return Max(Max(x, y), z);
}
```

Comme on le voit sur cet exemple, deux fonctions peuvent avoir le même nom, dès lors qu'elles diffèrent par le nombre ou le type des paramètres.

(2.6.5) **L'instruction switch**

Cette instruction permet un branchement conditionnel multiple.

Forme :

```
switch (<expression>)
{
        case <valeur1> :
                <instructions>
        case <valeur2> :
                <instructions>
            ⋮
        case <valeurn> :
                <instructions>
        default :
                <instructions>
}
```

Mécanisme : L'<*expression*> est évaluée. S'il y a un `case` avec une valeur égale à la valeur de l'<*expression*>, l'exécution est transférée à la première instruction qui suit ce `case` ; si un tel `case` n'existe pas, l'exécution est transférée à la première instruction qui suit `default :`.

Remarque 1.— La partie `default :` <*instructions*> est facultative.

Remarque 2.— L'instruction `switch` est en général utilisée en conjonction avec l'instruction `break;` qui permet de sortir immédiatement du `switch`.

Exemple :

```
switch (note / 2)             // on suppose note entier entre 0 et 20
{
        case 10 :
        case 9 :
        case 8 :
                mention = "TB";
                break;
        case 7 :
                mention = "B";
                break;
        case 6 :
                mention = "AB";
                break;
        case 5 :
                mention = "P";
                break;
        default :
                mention = "AJOURNE";
}
```

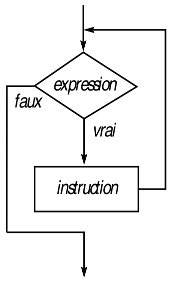
2.7 Boucles

(2.7.1) L'instruction while

Forme : `while` (<*expression entière*>)
<*instruction*>

Mécanisme : l'<*expression entière*> est d'abord évaluée. Tant qu'elle est vraie (c'est-à-dire $\neq 0$), l'<*instruction*> est effectuée.

Schématiquement :



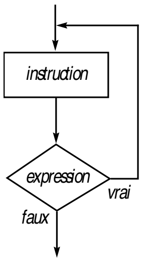
On remarque que le test est effectué avant la boucle.

(2.7.2) L'instruction do

Forme : `do`
<*instruction*>
`while` (<*expression entière*>);

Mécanisme : l'<*instruction*> est effectuée, puis l'<*expression entière*> est évaluée. Tant qu'elle est vraie (c'est-à-dire $\neq 0$), l'<*instruction*> est effectuée.

Schématiquement :



On remarque que le test est effectué après la boucle.

(2.7.3) **L'instruction for**

Forme :

```
for (<expression1>; <expression2>; <expression3>)
        <instruction>
```

Le mécanisme est équivalent à :

```
<expression1>;          // initialisation de la boucle
while (<expression2>)   // test de continuation de la boucle
{
        <instruction>
        <expression3>;  // évaluée à la fin de chaque boucle
}
```

Remarque 1.— Une boucle infinie peut se programmer ainsi :

```
for (;;)
        <instruction>
```

Remarque 2.— Dans une boucle, on peut utiliser les instructions :
`break;` pour abandonner immédiatement la boucle et passer à la suite,
`continue;` pour abandonner l'itération en cours et passer à la suivante,
mais l'usage de ces deux instructions se fait souvent au détriment de la clarté.

(2.7.4) **Exemple**

Ecrivons un bout de programme permettant d'entrer au clavier dix nombres entiers et qui affiche ensuite la moyenne arithmétique de ces dix nombres.

```
int n, somme = 0;
int i = 0;
cout << "Quand on le demande, tapez un entier suivi de <entrée>\n";
while (i < 10)          // boucle version while
{
        cout << "Entrée ? ";
        cin >> n;
        somme += n;
        i++;
}
cout << "La moyenne est " << somme / 10.0;
```

Ecrivons la même boucle avec un `do` :

```
do      // boucle version do
{
        cout << "Entrée ? ";
        cin >> n;
        somme += n;
        i++;
}
while (i < 10);
```

Avec un `for` :

```
for (i = 0; i < 10; i++)        // boucle version for
{
        cout << "Entrée ? ";
        cin >> n;
        somme += n;
}
```

(2.7.5) **Du bon choix d'une boucle**

Voici quelques principes qui pourront nous guider dans le choix d'une boucle :
- Si l'on sait combien de fois effectuer la boucle : utiliser `for`.
- Sinon, utiliser `while` ou `do` :
    - s'il y a des cas où l'on ne passe pas dans la boucle : utiliser `while`,
    - sinon, utiliser `do`.

Exemple : nous voulons définir une fonction calculant le pgcd de deux entiers > 0 en n'effectuant que des soustractions. Principe : soustraire le plus petit nombre du plus grand, et recommencer jusqu'à ce que les deux nombres soient égaux ; on obtient alors le pgcd. Par exemple, partant de 15 et 24, on obtient : $(15, 24) \to (15, 9) \to (6, 9) \to (6, 3) \to (3, 3)$ et le pgcd est 3. Connaissant les nombres a et b de départ, on ne peut pas savoir a priori combien d'itérations seront nécessaires. De plus si $a = b$, on a tout de suite le résultat, sans opération. Nous choisirons donc une boucle `while`, d'où :

```
int Pgcd(int a, int b)          // calcule le pgcd de a et b entiers positifs
{
        while (a != b)
                if (a > b)
                        a -= b;
                else
                        b -= a;
        return a;
}
```

2.8 Compléments

(2.8.1) **Expression virgule**

Forme : <*expression1*> **,** <*expression2*>

Fonctionnement :
- <*expression1*> est évaluée, sa valeur est oubliée
- <*expression2*> est ensuite évaluée et donne sa valeur à l'expression virgule.

Cette expression n'est intéressante que si <*expression1*> a un effet de bord.

(2.8.2) **Expression conditionnelle**

Forme : <*expression0*> `?` <*expression1*> `:` <*expression2*>

Fonctionnement : <*expression0*> est d'abord évaluée.

- si elle est non nulle, <*expression1*> est évaluée et donne sa valeur à l'expression conditionnelle,
- si elle est nulle, <*expression2*> est évaluée et donne sa valeur à l'expression conditionnelle.

Exemple :

```
int Pgcd(int a, int b) // calcule le pgcd de deux entiers positifs
                       // variante de (2.7.5) dans le style fonctionnel
{
        if (a == b)  return a;
        return  a > b ? Pgcd(a - b, b) : Pgcd(b - a, a);
}
```

(2.8.3) **Gestion de l'écran**

En mode texte, l'écran comporte 25 lignes et 80 colonnes.

`gotoxy(col, lig)`	place le curseur en colonne `col` et ligne `lig`,
`clreol()`	efface la fin de la ligne, sans déplacer le curseur,
`clrscr()`	efface l'écran.

Gestion du clavier

`kbhit()`	renvoie 1 (vrai) si on appuie sur une touche, 0 (faux) sinon
`getch()`	lit un caractère tapé, sans écho à l'écran, et le renvoie,
`getche()`	comme la précédente, mais avec écho à l'écran.

Pour être utilisées, ces fonctions nécessitent l'inclusion du fichier `conio.h` (voir (1.4.3)).

(2.8.4) **Nombres aléatoires**

`random(k)`	renvoie un entier aléatoire $\in \{0, \ldots, k-1\}$,
`randomize()`	initialise le générateur de nombres aléatoires

(à n'appeler qu'une seule fois, avant d'utiliser la fonction `random()`)

Pour être utilisées, ces fonctions nécessitent l'inclusion des fichiers `stdlib.h` et `time.h`.

(2.8.5) Signalons enfin la fonction :

`delay(n)` qui provoque une attente de n millièmes de secondes.

Son usage recquiert l'inclusion du fichier `dos.h`.

Chapitre 3

PROGRAMMATION DES CLASSES

3.1 Fonctions-membres

(3.1.1) L'objet implicite

Rappelons que pour décrire une classe (cf (1.2.5)), on commence par déclarer les données et fonctions-membres d'un objet de cette classe, puis on définit les fonctions-membres de ce même objet. Cet objet n'est jamais nommé, il est *implicite* (au besoin, on peut y faire référence en le désignant par `*this`).

Ainsi dans l'exemple du paragraphe (2.5.1), lorsqu'on écrit les définitions des fonctions-membres de la classe `Complexe`, on fait référence directement aux variables `re` et `im`, et ces variables sont les données-membres du nombre complexe implicite qu'on est en train de programmer et qui n'est jamais nommé. Mais s'il y a un autre nombre complexe, comme `g` dans la définition de la fonction `operator+`, les données-membres de l'objet `g` sont désignées par la notation pointée habituelle, à savoir `g.re` et `g.im`. Notons au passage que, bien que ces données soient privées, elles sont accessibles à ce niveau puisque nous sommes dans la définition de la classe `Complexe`.

(3.1.2) Flux de l'information

Chaque fonction-membre est une unité de traitement correspondant à une fonctionalité bien précise et qui sera commune à tous les objets de la classe.

Pour faire son travail lors d'un appel, cette unité de traitement dispose des informations suivantes :
- les valeurs des données-membre (publiques ou privées) de l'objet auquel elle appartient,
- les valeurs des paramètres qui lui sont transmises.

En retour, elle fournit un résultat qui pourra être utilisé après l'appel. Ainsi :

Avant de programmer une fonction-membre, il faudra identifier quelle est l'information qui doit y entrer (paramètres) et celle qui doit en sortir (résultat).

3.2 Constructeurs et destructeurs

(3.2.1) *Un constructeur est une fonction-membre déclarée du même nom que la classe, et sans type :*

`Nom_classe(`*<paramètres>*`);`

Fonctionnement : à l'exécution, l'appel au constructeur produit un nouvel objet de la classe, dont on peut prévoir l'initialisation des données-membres dans la définition du constructeur.

Exemple : avec la classe `Complexe` décrite en (2.5), l'expression `Complexe(1.0, 2.0)` a pour valeur un nombre complexe de partie réelle 1 et de partie imaginaire 2.

Dans une classe, il peut y avoir plusieurs constructeurs à condition qu'ils diffèrent par le nombre ou le type des paramètres. Un constructeur sans paramètre s'appelle *constructeur par défaut.*

(3.2.2) **Initialisation des objets**

Dans une classe, on peut ne pas mettre de constructeur. Dans ce cas, lors de la déclaration d'une variable de cette classe, l'espace mémoire est réservé mais les données-membres de l'objet ne reçoivent pas de valeur de départ : on dit qu'elles ne sont pas *initialisées*. Au besoin, on peut prévoir une fonction-membre publique pour faire cette initialisation. En revanche :

S'il y a un constructeur, il est automatiquement appelé lors de la déclaration d'une variable de la classe.

Exemples avec la classe `Complexe` déclarée en (2.5) :

```
Complexe z;             // appel automatique du constructeur par défaut
                        // équivaut à : Complexe z = Complexe();

Complexe z (1.0, 2.0);            // appel du constructeur paramétré
                        // équivaut à : Complexe z = Complexe(1.0, 2.0);
```

On retiendra que :

L'utilité principale du constructeur est d'effectuer des initialisations pour chaque objet nouvellement créé.

(3.2.3) **Initialisations en chaîne**

Si une classe `Class_A` contient des données-membres qui sont des objets d'une classe `Class_B`, par exemple :

```
class Class_A
{
public:
        Class_A(...);           // constructeur
        ...
private:
        Class_B  b1, b2;        // deux objets de la classe Class_B
        ...
};
```

alors, à la création d'un objet de la classe `Class_A`, le constructeur par défaut de `Class_B` (s'il existe) est automatiquement appelé pour chacun des objets `b1`, `b2` : on dit qu'il y a des *initialisations en chaîne*.

Mais pour ces initialisations, il est également possible de faire appel à un constructeur paramétré de `Class_B`, à condition de définir le constructeur de `Class_A` de la manière suivante :

```
Class_A :: Class_A(...)  :  b1 (...), b2 (...)
```
<instruction-bloc>

Dans ce cas, l'appel au constructeur de `Class_A` provoquera l'initialisation des données-membres `b1`, `b2` (par appel au constructeur paramétré de `Class_B`) *avant* l'exécution de l'*<instruction-bloc>*.

(3.2.4) Conversion de type

Supposons que `Ma_classe` comporte un constructeur à un paramètre de la forme :

```
Ma_classe(Mon_type x);
```

où `Mon_type` est un type quelconque.

Alors, chaque fois que le besoin s'en fait sentir, ce constructeur assure la conversion automatique d'une expression `e` de type `Mon_type` en un objet de type `Ma_classe` (à savoir `Ma_classe(e)`).

Par exemple, si nous avons dans la classe `Complexe` le constructeur suivant :

```
Complexe::Complexe(float x)
{
        re = x;
        im = 0.0;
}
```

alors ce constructeur assure la conversion automatique `float` → `Complexe`, ce qui nous permet d'écrire des instructions du genre :

```
z1 = 1.0;
z3 = z2 + 2.0;
```

(`z1`, `z2`, `z3` supposés de type `Complexe`).

(3.2.5) Destructeurs

Un destructeur est une fonction-membre déclarée du même nom que la classe mais précédé d'un tilda (~) et sans type ni paramètre :

```
~Nom_classe();
```

Fonctionnement : à l'issue de l'exécution d'un bloc, le destructeur est automatiquement appelé pour chaque objet de la classe `Nom_classe` déclaré dans ce bloc. Cela permet par exemple de programmer la restitution d'un environnement, en libérant un espace-mémoire alloué par l'objet. Nous n'en ferons pas souvent usage.

3.3 Surcharge des opérateurs

(3.3.1) En C++, on peut *surcharger* la plupart des opérateurs usuels du langage, c'est-à-dire les reprogrammer pour que, dans un certain contexte, ils fassent autre chose que ce qu'ils font d'habitude. Ainsi dans l'exemple (2.5), nous avons surchargé l'opérateur d'addition `+` pour pouvoir l'appliquer à deux nombres complexes et calculer leur somme.

Notons également que les opérateurs d'entrées-sorties `<<` et `>>` sont en réalité les surcharges de deux opérateurs de décalages de bits (appelés respectivement "shift left" et "shift right").

La surcharge d'un opérateur *<op>* se fait en déclarant, au sein d'une classe `Ma_classe`, une fonction-membre appelée `operator` *<op>*. Plusieurs cas peuvent se présenter, selon que *<op>* est un opérateur *unaire* (c'est-à-dire à un argument) ou *binaire* (c'est-a-dire à deux arguments). Nous allons voir quelques exemples.

(3.3.2) **Cas d'un opérateur unaire**

Nous voulons surcharger l'opérateur unaire `-` pour qu'il calcule l'opposé d'un nombre complexe. Dans la classe `Complexe` décrite en (2.5), nous déclarons la fonction-membre publique suivante :

```
Complexe operator-();
```

que nous définissons ensuite en utilisant le constructeur paramétré de la classe :

```
Complexe Complexe::operator-()
{
        return Complexe(-re, -im);
}
```

Par la suite, si `z` est une variable de type `Complexe`, on pourra écrire tout simplement l'expression `-z` pour désigner l'opposé de `z`, sachant que cette expression est équivalente à l'expression `z.operator-()` (message `operator-` destiné à `z`).

(3.3.3) **Cas d'un opérateur binaire**

Nous voulons surcharger l'opérateur binaire `-` pour qu'il calcule la différence de deux nombres complexes. Dans la même classe `Complexe`, nous déclarons la fonction-membre publique suivante :

```
Complexe operator-(Complexe u);
```

que nous définissons ensuite en utilisant également le constructeur paramétré de la classe :

```
Complexe Complexe::operator-(Complexe u)
{
        return Complexe(re - u.re, im - u.im);
}
```

Par la suite, si `z1` et `z2` sont deux variables de type `Complexe`, on pourra écrire tout simplement l'expression `z1 - z2` pour désigner le nombre complexe obtenu en soustrayant `z2` de `z1`, sachant que cette expression est équivalente à l'expression `z1.operator-(z2)` (message `operator-` destiné à `z1`, appliqué avec le paramètre d'entrée `z2`).

(3.3.4) **Autre cas d'un opérateur binaire**.

Cette fois, nous désirons définir un opérateur qui, appliqué à deux objets d'une même classe, donne une valeur d'un type différent. Ce cas est plus compliqué que le précédent.

On considère la classe "culinaire" suivante :

```
class Plat                  // décrit un plat proposé au menu d'un restaurant
{
public:
        float Getprix();    // fonction d'accès donnant le prix : voir (1.3.2)
private:
        char nom[20];       // nom du plat (voir chapitre 4)
        float prix;         // et son prix
}
```

Nous voulons surcharger l'opérateur `+` pour qu'en écrivant par exemple `poulet + fromage`, cela donne le prix total des deux plats (`poulet` et `fromage` supposés de type `Plat`).

Nous commençons par déclarer la fonction-membre :

```
float operator+(Plat p);
```

que nous définissons par :

```
float Plat::operator+(Plat p)
{
        return  prix + p.Getprix();
}
```

et que nous pouvons ensuite utiliser en écrivant par exemple `poulet + fromage`. Nous définissons ainsi une loi d'addition `+` : (`Plat` $\times$ `Plat`) $\rightarrow$ `float`.

Mais que se passe-t-il si nous voulons calculer `salade + poulet + fromage` ? Par associativité, cette expression peut également s'écrire :

```
salade + (poulet + fromage)
(salade + poulet) + fromage
```

donc il nous faut définir deux autres lois :

- une loi `+` : (`Plat` $\times$ `float`) $\rightarrow$ `float`,
- une loi `+` : (`float` $\times$ `Plat`) $\rightarrow$ `float`.

La première se programme en déclarant une nouvelle fonction-membre :

```
float operator+(float u);
```

que nous définissons par :

```
float Plat::operator+(float u)
{
        return  prix + u;
}
```

La deuxième ne peut pas se programmer avec une fonction-membre de la classe `Plat` puisqu'elle s'adresse à un `float`. Nous sommes contraints de déclarer une fonction libre :

```
float operator+(float u, Plat p);
```

que nous définissons par :

```
float operator+(float u, Plat p)
{
        return  u + p.Getprix();
}
```

3.4 Réalisation d'un programme

(3.4.1) Sur un exemple, nous allons détailler les différentes étapes qui mènent à la réalisation d'un programme. Il s'agira de simuler le jeu du "c'est plus, c'est moins" où un joueur tente de deviner un nombre choisi par le meneur de jeu.

Le fait de programmer avec des objets nous force à modéliser soigneusement notre application avant d'aborder le codage en C++.

(3.4.2) **1ère étape : identification des classes**

Conformément à (1.1.3), nous commençons par décrire le jeu de manière littérale :

> "Le jeu oppose un joueur à un meneur. Le meneur choisit un numéro secret (entre 1 et 100). Le joueur propose un nombre. Le meneur répond par : "c'est plus", "c'est moins" ou "c'est exact". Si le joueur trouve le numéro secret en six essais maximum, il gagne, sinon il perd."

Le jeu réunit deux acteurs avec des rôles différents : un meneur et un joueur. Nous définirons donc deux classes : une classe `Meneur` et une classe `Joueur`.

(3.4.3) **2ème étape : fiches descriptives des classes**

Il nous faut déterminer les données et fonctions-membres de chaque classe.

Le meneur détient un numéro secret. Ses actions sont :
- choisir ce numéro,
- répondre par un diagnostic ("c'est plus", "c'est moins" ou "c'est exact").

D'où la fiche descriptive suivante :

classe : `Meneur`	
privé :	*public :*
`numsecret`	`Choisis`
	`Reponds`

Le joueur détient un nombre (sa proposition). Son unique action est de proposer ce nombre. Mais au cours du jeu, il doit garder à l'esprit une fourchette dans laquelle se situe le numéro à deviner, ce qui nous amène à la fiche descriptive suivante :

classe : `Joueur`	
privé :	*public :*
`proposition`	`Propose`
`min`	
`max`	

(3.4.4) $3^{\text{ème}}$ étape : description détaillée des fonctions-membres

Nous allons décrire le fonctionnement de chaque fonction-membre et en préciser les informations d'entrée et de sortie (voir (3.1.2)).

`Choisis` (de `Meneur`) :
- entrée : rien
- sortie : rien
- choisit la valeur de `numsecret`, entre 1 et 100. On remarque que ce choix ne se fait qu'une fois, au début de la partie. Il est donc logique que ce soit le constructeur qui s'en charge. Nous transformerons donc cette fonction en constructeur.

`Reponds` (de `Meneur`) :
- entrée : la proposition du joueur
- sortie : un diagnostic : "exact", "plus" ou "moins" (que nous coderons respectivement par 0, 1 ou 2)
- compare la proposition du joueur avec le numéro secret et rend son diagnostic.

`Propose` (de `Joueur`) :
- entrée : le diagnostic du précédent essai
- sortie : un nombre
- compte tenu des tentatives précédentes, émet une nouvelle proposition.

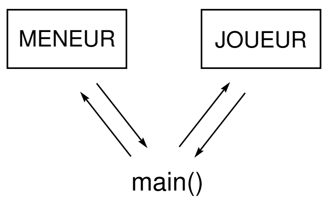
(3.4.5) $4^{\text{ème}}$ étape : description du traitement principal

La fonction `main()` sera le chef-d'orchestre de la simulation. Son travail consiste à :

- déclarer un joueur et un meneur
- faire :
  - prendre la proposition du joueur
  - la transmettre au meneur
  - prendre le diagnostic du meneur
  - le transmettre au joueur

  jusqu'à la fin de la partie
- afficher le résultat

Remarquons que nos deux objets-acteurs ne communiquent entre eux que de manière indirecte, par l'intermédiaire de la fonction `main()` :



Nous verrons plus tard comment mettre directement en rapport les objets entre eux, à l'aide des pointeurs.

(3.4.6) $5^{\text{ème}}$ étape : déclaration des classes

Nous en arrivons à la programmation proprement dite. Nous commençons par écrire les fichiers de déclarations des classes `Meneur` et `Joueur` :

```
// ----------------------- meneur.h -----------------------
// ce fichier contient la déclaration de la classe Meneur

class Meneur
{
public:
    Meneur();                   // initialise un meneur
    int Reponds(int prop);      // reçoit la proposition du joueur
                                // renvoie 0 si c'est exact, 1 si c'est plus
                                // et 2 si c'est moins
private:
    int numsecret;              // numéro secret choisi au départ
};

// ----------------------- joueur.h -----------------------
// ce fichier contient la déclaration de la classe Joueur

class Joueur
{
public:
    Joueur();                   // initialise un joueur
    int Propose(int diag);      // reçoit le diagnostic du précédent essai
                                // renvoie une nouvelle proposition
private:
    int min, max,               // fourchette pour la recherche
        proposition;            // dernier nombre proposé
};
```

(3.4.7) **6ème étape : écriture du traitement principal**

Ce fichier contient l'utilisation des classes.

```
// ----------------------- jeu.cpp --------------------
// programme de simulation du jeu "c'est plus, c'est moins"

#include <iostream.h>           // pour les entrées-sorties
#include "joueur.h"             // pour la déclaration de la classe Joueur
#include "meneur.h"             // pour la déclaration de la classe Meneur

void main()                     // gère une partie ...
{
    Joueur j;                   // ...  avec un joueur ...
    Meneur m;                   // ...  et un meneur
    int p, d = 1,               // variables auxiliaires
        cpt = 0;                // nombre d'essais
    do                          // simulation du déroulement du jeu
    {
        p = j.Propose(d);       // proposition du joueur
        d = m.Reponds(p);       // diagnostic du meneur
        cpt++;
    }
    while (d && cpt < 6);
    if (d)                      // défaite du joueur
        cout << "\nLe joueur a perdu !";
    else                        // victoire du joueur
        cout << "\nLe joueur a gagné !";
}
```

Nous pouvons dès à présent compiler ce fichier `jeu.cpp`, alors que les classes `Joueur` et `Meneur` ne sont pas encore définies.

(3.4.8) 7$^{\text{ème}}$ étape : définition des classes

C'est l'ultime étape, pour laquelle nous envisagerons deux scénarios différents. Dans le premier, l'utilisateur du programme tiendra le rôle du joueur tandis que l'ordinateur tiendra le rôle du meneur. Dans le second, ce sera le contraire : l'utilisateur tiendra le rôle du meneur et l'ordinateur celui du joueur. Nous écrirons donc deux versions des classes `Meneur` et `Joueur`.

Première version :

```
// ----------------------- meneur.cpp ---------------------
// ce fichier contient la définition de la classe Meneur
// le rôle du meneur est tenu par l'ordinateur

#include <iostream.h>
#include <stdlib.h>                   // pour random()
#include <time.h>                     // et randomize()
#include "meneur.h"

Meneur::Meneur()
{
    randomize();                      // initialisation du générateur aléatoire
    numsecret = 1 + random(100);      // choix du numéro secret
}

int Meneur::Reponds(int prop)         // prop = proposition du joueur
{
    if (prop < numsecret)
    {
        cout << "\nC'est plus";
        return 1;
    }
    if (prop > numsecret)
    {
        cout << "\nC'est moins";
        return 2;
    }
    cout << " \nC'est exact";
    return 0;
}

// ----------------------- joueur.cpp ---------------------
// ce fichier contient la définition de la classe Joueur
// le rôle du joueur est tenu par l'utilisateur du programme

#include <iostream.h>
#include "joueur.h"

Joueur::Joueur()
{
    cout << "\nBonjour !  Vous allez jouer le rôle du joueur.";
}

int Joueur::Propose(int diag)         // la valeur de diag est ignorée
{
    int p;
    cout << "\nProposition ?  ";
    cin >> p;
    return p;
}
```

Seconde version :

```
// ----------------------- meneur.cpp --------------------
// ce fichier contient la définition de la classe Meneur
// le rôle du meneur est tenu par l'utilisateur du programme

#include <iostream.h>
#include "meneur.h"

Meneur::Meneur()
{
    cout << "\nBonjour !  Vous allez jouer le rôle du meneur.";
    cout << "\nChoisissez un numéro secret entre 1 et 100";
}

int Meneur::Reponds(int prop)         // la valeur de prop est ignorée
{
    int r;
    cout << "\n0 - C'est exact";
    cout << "\n1 - C'est plus";
    cout << "\n2 - C'est moins";
    cout << "\nVotre réponse (0,1,2) ?  ";
    cin >> r;
    return r;
}

// ----------------------- joueur.cpp --------------------
// ce fichier contient la définition de la classe Joueur
// le rôle du joueur est tenu par l'ordinateur

#include <iostream.h>
#include "joueur.h"

Joueur::Joueur()
{
    min = 1;                          // fixe la fourchette dans laquelle
    max = 100;                        // se trouve le numéro à deviner
    proposition = 0;                  // première proposition fictive
}

int Joueur::Propose(int diag)
{
    if (diag == 1)                    // ajuste la fourchette
        min = proposition + 1;
    else
        max = proposition - 1;
    proposition = (min + max) / 2;    // procède par dichotomie
    cout << "\nJe propose :  " << proposition;
    return proposition;
}
```

Remarque.— Quand nous aurons abordé les notions d'héritage et de polymorphisme, nous serons en mesure d'écrire une version unique et beaucoup plus souple de ce programme, où la distribution des rôles pourra être décidée au moment de l'exécution.

Chapitre 4

LES INGREDIENTS DES CLASSES
(suite)

4.1 Enumérations

(4.1.1) *Les énumérations sont des listes de noms représentant les valeurs entières successives* $0, 1, 2, \ldots$

Une énumération se définit par un énoncé de la forme :

`enum` <*nom*> { <*liste de noms*> }`;`

Exemples :

```
enum Jour {dimanche, lundi, mardi, mercredi, jeudi, vendredi, samedi};
enum Couleur {coeur, pique, carreau, trefle};
enum Vache {blanchette, noiraude, gertrude};
```

Le premier énoncé équivaut à la déclaration des constantes entières :

```
const int dimanche = 0;
const int lundi = 1;
const int mardi = 2;
etc ...
```

et attribue à ces constantes un type appelé `Jour`.

(4.1.2) On pourra utiliser les énumérations pour déclarer des variables, comme par exemple :

```
Jour fatal;
Couleur c;
Vache folle = gertrude;
```

4.2 Tableaux

(4.2.1) *Un tableau est une collection indicée de variables de même type.*

Forme de la déclaration :

<*type*> <*nom*> `[`<*taille*>`];`

où : <*type*> est le type des éléments du tableau,
<*nom*> est le nom du tableau,
<*taille*> est une constante entière égale au nombre d'éléments du tableau.

Exemples :

```
int fraction[2];              // tableau de deux entiers
char mot[10], s[256];         // tableaux de respectivement 10 et 256 caractères
Complexe tab[3];              // tableau de trois objets de type Complexe
Vache troupeau[1000];         // tableau de mille vaches
```

(4.2.2) *Si* `t` *est un tableau et* `i` *une expression entière, on note* `t[i]` *l'élément d'indice* `i` *du tableau. Les éléments d'un tableau sont indicés de* 0 *à* $taille - 1$.

Par exemple, le tableau `fraction` déclaré ci-dessus représente deux variables entières qui sont `fraction[0]` et `fraction[1]`. Ces variables se traitent comme des variables ordinaires ; on peut par exemple écrire :

```
fraction[0] = 1;
fraction[1] = 2;   etc.
```

Remarque.— Les éléments d'un tableau sont stockés en mémoire de façon contiguë, dans l'ordre des indices.

(4.2.3) Il est possible de déclarer des tableaux à deux indices (ou plus). Par exemple, en écrivant :

```
int M[2][3];
```

on déclare un tableau d'entiers `M` à deux indices, le premier indice variant entre 0 et 1, le second entre 0 et 2. On peut voir `M` comme une matrice d'entiers à 2 lignes et 3 colonnes. Les éléments de `M` se notent `M[i][j]`.

(4.2.4) Comme pour les variables ordinaires (cf (2.1.5)), on peut faire des déclarations-initialisations de tableaux. En voici trois exemples :

```
int T[3] = {5, 10, 15};
char voyelle[6] = {'a', 'e', 'i', 'o', 'u', 'y'};
int M[2][3] = {{1, 2, 3}, {3, 4, 5}};   // ou : int M[2][3] = {1, 2, 3, 3, 4, 5};
```

(4.2.5) **Déclaration typedef**

Elle permet d'attribuer un nom à un type.

Forme générale : `typedef` <*déclaration*>

où <*déclaration*> est identique à une déclaration de variable, dans laquelle le rôle du nom de la variable est joué par le nom du type que l'on veut définir.

C'est très pratique pour nommer certains types de tableaux. Exemples :

```
const int DIM = 3;
typedef float Vecteur[DIM];    // définit un type Vecteur (= tableau de trois réels)
typedef int Matrice[2][3];     // définit un type Matrice
typedef char Phrase[256];      // définit un type Phrase
```

Cela permet de déclarer ensuite, par exemple :

```
Vecteur U, V = {0.0, 0.0, 0.0}, W;
Matrice M;      // déclaration équivalente à celle de (4.2.3)
Phrase s;
```

(4.2.6) **Utilisation des tableaux**

On retiendra les trois règles suivantes :

- *Les tableaux peuvent être passés en paramètres dans les fonctions,*
- *les tableaux ne peuvent être renvoyés (avec* `return`*) comme résultats de fonctions,*
- *l'affectation entre tableaux est interdite.*

Exemple 1.— Instruction qui copie le vecteur `U` dans `V` :

```
for (int i = 0; i < DIM; i++)
        V[i] = U[i];
```

Exemple 2.— Fonction qui affiche un vecteur :

```
void Affiche(Vecteur V)        // affiche les composantes du vecteur V
{
        for (int i = 0; i < DIM; i++)
                cout << V[i] << " ";
}
```

Exemple 3.— Fonction renvoyant un vecteur nul :

```
Vecteur Zero()
{
        Vecteur Z;
        for (int i = 0; i < DIM; i++)
                Z[i] = 0.0;
        return Z;
}
```

ILLEGAL !

(4.2.7) **Tableaux d'objets et constructeurs**

Supposons que l'on déclare un tableau d'objets, par exemple :

```
Complexe tab[3];
```

La variable `tab` n'est pas un objet. A sa création, le constructeur par défaut de la classe `Complexe` (s'il existe) est appelé automatiquement pour chaque élément du tableau. La déclaration ci-dessus est équivalente à :

```
Complexe tab[3] = {Complexe(), Complexe(), Complexe()};
```

Il est également possible d'invoquer un constructeur paramétré de la classe `Complexe` en écrivant par exemple :

```
Complexe tab[3] = {Complexe(), Complexe(1.0, 0.0), Complexe(0.0, 1.0)};
```

4.3 Chaînes de caractères

(4.3.1) *En C++, une chaîne de caractères n'est rien d'autre qu'un tableau de caractères, avec un caractère nul* `'\0'` *marquant la fin de la chaîne.*

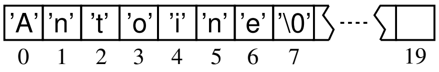
Exemples de déclarations :

```
char nom[20], prenom[20];       // 19 caractères utiles
char adresse[3][40];            // trois lignes de 39 caractères utiles
char turlu[10] = {'t', 'u', 't', 'u', '\0'}; // ou :  char turlu[10] = "tutu";
```

(4.3.2) Comme il a été dit en (2.1.3), les valeurs explicites de type chaîne de caractères sont écrites avec des guillemets. Par exemple :

```
prenom = "Antoine";             // illégal d'après (4.2.6)
strcpy(prenom, "Antoine");      // correct :  voir ci-dessous (4.3.3)
```

On peut représenter le tableau `prenom` par le schéma suivant :



Le nombre de caractères précédant `'\0'` s'appelle la *longueur* de la chaîne.

Les chaînes de caractère peuvent être saisies au clavier et affichées à l'écran grâce aux objets habituels `cin` et `cout`.

(4.3.3) Dans la librairie standard du C++, il y a de nombreuses fonctions utilitaires sur les chaînes de caractères. Parmi elles :

`strlen(s)` (dans `string.h`) : donne la longueur de la chaîne `s`
`strcpy(dest, source)` (dans `string.h`) : recopie la chaîne `source` dans `dest`
`strcmp(s1,s2)` (dans `string.h`) : compare les chaînes `s1` et `s2` :
    renvoie une valeur < 0, nulle ou < 0 selon que `s1` est inférieure, égale ou supérieure à `s2` pour l'ordre alphabétique.
`gets(s)` (dans `stdio.h`) :
    lit une chaîne de caractères tapée au clavier, jusqu'au premier retour à la ligne (inclus) ; les caractères lus sont rangés dans la chaîne `s`, sauf le retour à la ligne qui y est remplacé par `'\0'`.

(4.3.4) **Utilisation des chaînes de caractères**

Nous en donnerons deux exemples.

Exemple 1.— Fonction qui indique si une phrase est un palindrome (c'est-à-dire qu'on peut la lire aussi bien à l'endroit qu'à l'envers) :

```
int Palindrome(Phrase s)        // teste si s est un palindrome
{
        int i = 0, j = strlen(s) - 1;
        while (s[i] == s[j] && i < j)
                i++, j--;
        return  i >= j;
}
```

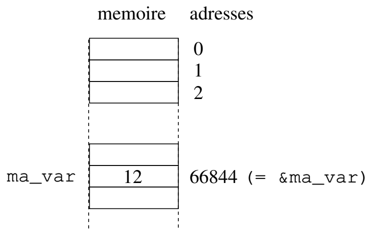
Exemple 2.— Instructions qui, à partir d'une chaîne `s`, fabrique la chaîne `t` obtenue en renversant `s` :

```
for (int i = 0, j = strlen(s) - 1; i < strlen(s); i++, j--)
        t[i] = s[j];
t[strlen(s)] = '\0';
```

4.4 Pointeurs

(4.4.1) **Adresses**

La mémoire d'un ordinateur peut être considérée comme un empilement de cases-mémoire. Chaque case-mémoire est repérée par un numéro qu'on appelle son *adresse* (en général un entier long). Si `ma_var` est une variable d'un type quelconque, l'adresse où est stockée la valeur de `ma_var` s'obtient grâce à l'*opérateur d'adresse* `&` : cette adresse se note `&ma_var` :



(4.4.2) **Pointeurs**

Un pointeur est une variable qui contient l'adresse d'une autre variable.

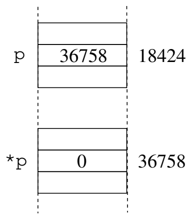
La déclaration d'un pointeur est de la forme suivante :

<*type*> `*`<*nom*>`;`

Par exemple :

```
int *p;
```

La variable `p` est alors un *pointeur sur un* `int` ; la variable entière dont `p` contient l'adresse est dite *pointée par* `p` et se note `*p`. Schématiquement :



Une variable pointée s'utilise comme un variable ordinaire. Exemple :

```
int *p, *q, n;
n = -16;
*p = 101;
*q = n + *p;    // donne à *q la valeur 85
```

(4.4.3) Un pointeur est toujours lié à un type (sur lequel il pointe). Exemple :

```
int *ip;
double *dp;
dp = ip;        // illégal car ip et dp ne sont pas de même type
*dp = *ip;      // correct (conversion implicite de type)
```

Remarque.— Il y a dans `stdlib.h` la valeur pointeur `NULL` (égale à 0) qui est compatible avec tout type de pointeur. Ainsi :

```
dp = NULL;      // légal
ip = NULL;      // légal
```

(4.4.4) **Références**

Une référence est une variable qui coïncide avec une autre variable.

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

<type> `&`*<nom>* `=` *<nom_var>*`;`

Par exemple :

```
int n = 10;
int &r = n;     // r est une référence sur n
```

Cela signifie que `r` et `n` représentent la même variable (en particulier, elles ont la même adresse). Si on écrit par la suite : `r = 20`, `n` prend également la valeur 20.

Attention :

```
double somme, moyenne;
double &total = somme;          // correct :  total est une référence sur somme
double &valeur;                 // illégal :  pas de variable à laquelle se référer
double &total = moyenne;        // illégal :  on ne peut redéfinir une référence
```

(4.4.5) **Passage des paramètres dans une fonction**

Supposons que nous voulions définir une fonction `echange` permettant d'échanger les valeurs de deux variables entières.

- Première version, naïve :

```
void echange(int a, int b)      // version erronée
{
        int aux;                // variable auxiliaire servant pour l'échange
        aux = a; a = b; b = aux;
}
```

Ici l'expression `echange(x, y)` (où `x` et `y` sont de type `int`) n'a aucun effet sur `x` et `y`. En effet, comme il a été dit en (2.4.4), lors de l'appel de la fonction, les valeurs de `x` et `y` sont recopiées dans les variables locales `a` et `b` (on parle de *passage par valeur*) et ce sont les valeurs de ces variables `a` et `b` qui sont échangées.

- Deuxième version, avec des références :

```
void echange(int &a, int &b)    // version correcte, style C++
{
        int aux;
        aux = a; a = b; b = aux;
}
```

Cette fois, l'expression `echange(x, y)` réalise correctement l'échange des valeurs de `x` et `y`, car les variables `a` et `b` coïncident avec les variables `x` et `y` (on parle de *passage par référence*).

- Troisième version moins pratique, avec des pointeurs :

```
void echange(int *a, int *b)    // version correcte, style C
{
        int aux;
        aux = *a; *a = *b; *b = aux;
}
```

Pour échanger les valeurs de x et y, il faut alors écrire `echange(&x, &y)`, c'est-à-dire passer à la fonction les adresses de x et y (on parle de *passage par adresse*).

(4.4.6) **Pointeurs et tableaux**

Considérons la déclaration suivante :

```
int T[5];       // T est un tableau de 5 entiers
```

`T` est en réalité un *pointeur constant sur un* `int` et contient l'adresse du premier élément du tableau. Donc `T` et `&T[0]` sont synonymes. On a le droit d'écrire par exemple :

```
int *p = T;
p[2];           // = T[2]
```

Remarque.— On peut ainsi faire ce qu'on appelle de *l'arithmétique sur les pointeurs*. Par exemple :

`p + i` pointe sur l'élément d'indice `i` du tableau,
`*(p+i)` est la même chose que `p[i]` ou `T[i]`.

A noter :

```
int U[5];
U = T;          // illégal : U ne peut être modifié (pointeur constant) : voir (4.2.6)
```

(4.4.7) Comme conséquence de (4.4.6), on voit que, comme paramètres de fonctions, les tableaux passent par adresse. Donc :

- il n'y a pas de recopie du contenu du tableau (d'où gain de temps),
- la fonction peut modifier le contenu du tableau.

Par exemple, en définissant :

```
void annule(int V[])    // pour un paramètre tableau : inutile d'indiquer la taille
{
        for (int i = 0; i < 5; i++)
                V[i] = 0;
}
```

on peut mettre à zéro un tableau de cinq entiers `T` en écrivant `annule(T)`.

(4.4.8) **Cas des chaînes de caractères**

Une chaîne de caractères est habituellement déclarée :

- soit comme tableau de caractères, comme en (4.3) ; exemple : `char nom[10];`
- soit comme pointeur sur un caractère ; exemple : `char *nom;`

Avec la première déclaration, l'espace-mémoire est automatiquement réservé pour recevoir 10 caractères. Avec la deuxième, aucun espace n'est a priori réservé : il faudra faire cette réservation nous-même (voir paragraphe suivant).

D'autre part, avec la deuxième déclaration, le pointeur `nom` peut être modifié. Avec la première, il ne le peut pas.

A titre d'exemple, voici deux définitions équivalentes de la fonction `strcpy` mentionnée au paragraphe (4.3.3). Cette fonction recopie une chaîne de caractères `source` dans une chaîne `dest` (attention : l'expression `dest = source` recopierait le pointeur mais pas la chaîne !) :

```
char *strcpy(char *dest, const char *source)          // copie source dans dest
{
        for (int i = 0; source[i] != '\0'; i++)
                dest[i] = source[i];
        dest[i] = source[i];           // copie le terminateur '\0'
        return dest;
}

char *strcpy(char *dest, const char *source)          // copie source dans dest
{
        char *temp = dest;
        while (*source)
                *dest++ = *source++;
        *dest = *source;               // copie le terminateur '\0'
        return temp;
}
```

(4.4.9) **Gestion de la mémoire — variables dynamiques**

a) Supposons déclaré :

```
int *p;         // p :  pointeur sur int
```

Pour l'instant, `p` ne pointe sur rien. Il faut réserver l'espace-mémoire nécessaire à un `int` par l'instruction :

```
p = new int;    // new :  opérateur d'allocation mémoire
```

Cette fois, `p` contient une véritable adresse (ou `NULL` s'il n'y a plus assez de mémoire). On peut alors utiliser la variable pointée par `p` (qu'on nomme *variable dynamique* par opposition aux variables ordinaires qualifiées de *statiques*). Par exemple :

```
*p = 12;        // ou tout autre traitement...
```

Lorsqu'on n'a plus besoin de la variable `*p`, il faut libérer l'espace-mémoire qu'elle occupe par l'instruction :

```
delete p;       // delete :  opérateur de désallocation mémoire
```

L'expression `*p` est dès lors illégale, jusqu'à une prochaine allocation par : `p = new...`.

b) *(cas d'un tableau dynamique)* Supposons déclaré :

```
int *T;
```

L'allocation-mémoire nécessaire pour recevoir n entiers s'écrira :

```
T = new int [n];        // T est maintenant un tableau de n entiers
```

Ici, l'intérêt est que `n` peut être n'importe quelle expression entière, et pas seulement une constante comme dans la déclaration d'un tableau statique (voir (4.2.1)). La taille du tableau peut donc être décidée au moment de l'exécution, en fonction des besoins.

En fin de traitement, la libération de l'espace-mémoire occupé par `T` s'écrit :

```
delete [] T;            // désallocation mémoire d'un tableau dynamique
```

A retenir :

Grâce aux pointeurs, on peut manipuler des structures de données dynamiques dont la taille n'est déterminée qu'au moment de l'exécution.

c) L'espace-mémoire (nombre d'octets) occupé par une valeur de type `un_type` est donné par l'opérateur `sizeof` : on écrira `sizeof un_type` ou encore `sizeof une_expr`, où `une_expr` est une expression de type `un_type`.

(4.4.10) **Pointeurs et objets**

a) Ce qui précède s'applique également aux types classes. Nous pourrons utiliser des objets dynamiques, par exemple de la classe `Complexe` (cf (2.5)), en écrivant :

```
Complexe *pc = new Complexe;   // etc...
```

De la même manière qu'avec les objets statiques :

- si `new` est utilisé avec un type classe, le constructeur par défaut (s'il existe) est appelé automatiquement,
- il est possible de faire un appel explicite à un constructeur, par exemple :

```
Complexe *pc = new Complexe(1.0, 2.0);
```

b) L'accès aux données et fonctions-membres d'un objet pointé peut se faire grâce à l'*opérateur flèche* `->`. Par exemple, avec les déclarations du a), on écrira :

```
pc -> re              plutôt que   (*pc).re
pc -> Affiche()       plutôt que   (*pc).Affiche()
```

(4.4.11) **Liens entre objets**

Supposons déclarées deux classes avec :

```
class A
{
public:
        machin truc();  // fonction-membre
.....
};

class B
{
private:
        A *pa;          // donnée-membre :  pointeur sur un objet de la classe A
.....
};
```

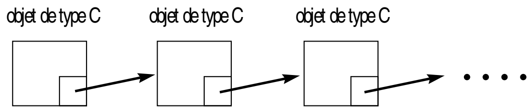
Comme un objet `obj_B` de la classe `B` contient un pointeur sur un objet de la classe `A`, cela permet à `obj_B` de communiquer directement avec cet objet en lui envoyant par exemple le message `pa -> truc()`. Ainsi :

En programmation-objet, l'utilité principale des pointeurs est de permettre à deux objets de communiquer directement entre eux.

On peut également construire des *listes chaînées* d'objets de même classe, en déclarant :

```
class C
{
private:
        C *lien;        // donnée-membre :  pointeur sur un objet de la classe C
.....
};
```

Une telle liste peut être représentée par le schéma suivant :



4.5 Récapitulatif des opérateurs

Voici un tableau des opérateurs du langage C++, classés par niveaux de priorité décroissante. Pour un même niveau, l'évaluation se fait de gauche à droite ou de droite à gauche selon le sens de l'associativité.

<table>
<tr><th>Priorité</th><th>Opérateurs</th><th>Associativité</th></tr>
<tr><td>1</td><td><code>()</code> (appel de fonction) <code>[]</code> <code>-></code> <code>::</code> <code>.</code></td><td>gauche à droite</td></tr>
<tr><td>2</td><td><code>!</code> <code>~</code>(négation bit à bit) <code>+</code> <code>-</code> <code>++</code> <code>--</code> <code>&</code> <code>*</code> (opérateurs unaires)
<code>sizeof</code> <code>new</code> <code>delete</code> <code>()</code> (conversion de type)</td><td>droite à gauche</td></tr>
<tr><td>3</td><td><code>.*</code> <code>->*</code></td><td>gauche à droite</td></tr>
<tr><td>4</td><td><code>*</code> (opérateur binaire) <code>/</code> <code>%</code></td><td>gauche à droite</td></tr>
<tr><td>5</td><td><code>+</code> <code>-</code> (opérateurs binaires)</td><td>gauche à droite</td></tr>
<tr><td>6</td><td><code><<</code> <code>>></code> (décalages de bits)</td><td>gauche à droite</td></tr>
<tr><td>7</td><td><code><</code> <code><=</code> <code>></code> <code>>=</code></td><td>gauche à droite</td></tr>
<tr><td>8</td><td><code>==</code> <code>!=</code></td><td>gauche à droite</td></tr>
<tr><td>9</td><td><code>&</code> ("et" bit à bit)</td><td>gauche à droite</td></tr>
<tr><td>10</td><td><code>^</code> ("ou" exclusif bit à bit)</td><td>gauche à droite</td></tr>
<tr><td>11</td><td><code>|</code> ("ou" bit à bit)</td><td>gauche à droite</td></tr>
<tr><td>12</td><td><code>&&</code></td><td>gauche à droite</td></tr>
<tr><td>13</td><td><code>||</code></td><td>gauche à droite</td></tr>
<tr><td>14</td><td><code>?:</code></td><td>droite à gauche</td></tr>
<tr><td>15</td><td><code>=</code> <code>*=</code> <code>/=</code> <code>%=</code> <code>+=</code> <code>-=</code> <code>&=</code> <code>^=</code> <code>|=</code> <code><<=</code> <code>>>=</code></td><td>droite à gauche</td></tr>
<tr><td>16</td><td><code>,</code></td><td>gauche à droite</td></tr>
</table>

Seuls cinq opérateurs ne peuvent être surchargés ; ce sont : `.` `.*` `::` `?:` `sizeof`

Chapitre 5

HERITAGE

5.1 Relations *a-un*, *est-un*, *utilise-un*

(5.1.1) Dans le cadre d'un programme concernant les transports, supposons que nous déclarions les quatre classes suivantes : `Voiture`, `Moteur`, `Route` et `Vehicule`. Quel type de relation y a-t-il entre la classe `Voiture` et les trois autres classes ?

Tout d'abord, on peut dire qu'une voiture *a un* moteur : le moteur fait partie de la voiture, il est contenu dans celle-ci.

Ensuite, une voiture *utilise une* route, mais il n'y a pas d'inclusion : la route ne fait pas partie de la voiture, de même que la voiture ne fait pas partie de la route.

Enfin, une voiture *est un* véhicule, d'un genre particulier : la voiture possède toutes les caractéristiques d'un véhicule, plus certaines caractéristiques qui lui sont propres.

(5.1.2) Du point de vue de la programmation, la relation *a-un* est une inclusion entre classes. Ainsi, un objet de type `Voiture` renferme une donnée-membre qui est un objet de type `Moteur`.

La relation *utilise-un* est une collaboration entre classes indépendantes. Elle se traduit le plus souvent par des pointeurs. Par exemple, un objet de type `Voiture` renfermera une donnée-membre de type pointeur sur `Route`, ce qui permettra à la voiture de communiquer avec une route (cf. (4.4.11)).

La relation *est-un* s'appelle un *héritage* : une voiture hérite des caractéristiques communes à tout véhicule. On dira que la classe `Voiture` est *dérivée* de la classe `Vehicule`.

5.2 Classes dérivées

(5.2.1) Le principe est d'utiliser la déclaration d'une classe — appelée *classe de base* ou classe *parente* — comme base pour déclarer une seconde classe — appelée *classe dérivée*. La classe dérivée héritera de tous les membres (données et fonctions) de la classe de base.

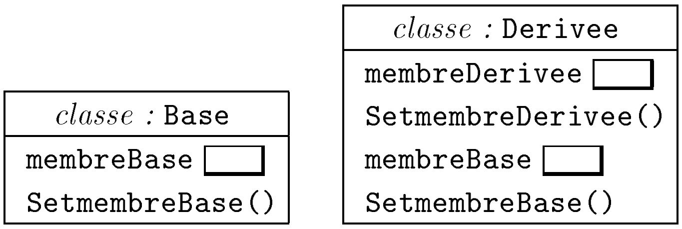
Considérons par exemple la déclaration suivante :

```
class Base
{
public:
        short membreBase;
        void SetmembreBase(short valeurBase);
};
```

On déclare une classe dérivée de la classe `Base` grâce au qualificatif `public Base`. Par exemple :

```
class Derivee :  public Base           // héritage public
{
public:
        short membreDerivee;
        void SetmembreDerivee(short valeurDerivee);
};
```

Un objet de la classe `Derivee` possède alors ses propres données et fonctions-membres, plus les données-membres et fonctions-membres héritées de la classe `Base` :



(5.2.2) **Contrôle des accès**

Il est possible de réserver l'accès à certaines données (ou fonctions) membres de la classe de base aux seules classes dérivées en leur mettant le qualificatif `protected:`

A retenir :

Les données et fonctions-membres privées sont inaccessibles aux classes dérivées.

(5.2.3) **Premier exemple**

```
# include <iostream.h>

class Base
{
public:
        void SetmembreBase(short valeurBase);
protected:
        short membreBase;
};

void Base::SetmembreBase(short valeurBase)
{
        membreBase = valeurBase;
}

class Derivee :  public Base
{
public:
        void SetmembreDerivee(short valeurDerivee);
        void AfficheDonneesMembres(void);
private:
        short membreDerivee;
};
```

```
void Derivee::SetmembreDerivee(short valeurDerivee)
{
        membreDerivee = valeurDerivee;
}

void Derivee::AfficheDonneesMembres(void)
{
        cout << "Le membre de Base a la valeur " << membreBase << "\n";
        cout << "Le membre de Derivee a la valeur " << membreDerivee << "\n";
}

void main()
{
        Derivee *ptrDerivee;
        ptrDerivee = new Derivee;
        ptrDerivee -> SetmembreBase(10);              // message 1
        ptrDerivee -> SetmembreDerivee(20);           // message 2
        ptrDerivee -> AfficheDonneesMembres();        // message 3
}
```

A l'exécution, ce programme affichera les deux lignes suivantes :

```
Le membre de Base a la valeur 10
Le membre de Derivee a la valeur 20
```

(5.2.4) Héritage public ou privé

Il est possible de déclarer :

```
        class Derivee :  public Base          // héritage public
ou :    class Derivee :  private Base         // héritage privé
```

Il faut savoir que :

- *dans le premier cas, les membres hérités conservent les mêmes droits d'accès (*`public` *ou* `protected`*) que dans la classe de base,*
- *dans le second cas (cas par défaut si rien n'est précisé), tous les membres hérités deviennent privés dans la classe dérivée.*

On conseille généralement d'utiliser l'héritage public, car dans le cas contraire on se prive de pouvoir créer de nouvelles classes elles-mêmes dérivées de la classe dérivée.

(5.2.5) Constructeurs et destructeurs

Quand un objet est créé, si cet objet appartient à une classe dérivée, le constructeur de la classe parente est *d'abord* appelé. Quand un objet est détruit, si cet objet appartient à une classe dérivée, le destructeur de la classe parente est appelé *après*.

Ces mécanismes se généralisent à une chaîne d'héritages. Voici un exemple :

```
# include <iostream.h>

class GrandPere
{
        .....  // données-membres
public:
        GrandPere(void);
        ~GrandPere();
};
```

```
class Pere : public GrandPere
{
        .....  // données-membres
public:
        Pere(void);
        ~Pere();
};

class Fils : public Pere
{
        .....  // données-membres
public:
        Fils(void);
        ~Fils();
};

void main()
{
        Fils *junior;
        junior = new Fils;
                // appels successifs des constructeurs de GrandPere, Pere et Fils
        ........
        delete junior;
                // appels successifs des destructeurs de Fils, Pere et GrandPere
}
```

(5.2.6) Cas des constructeurs paramétrés

Supposons déclarées les classes suivantes :

```
class Base
{
        .....
        Base(short val);
};

class Derivee : public Base
{
        .....
        Derivee(float x);
};
```

Dans la définition du constructeur de `Derivee`, on pourra indiquer quelle valeur passer au constructeur de la classe `Base` de la manière suivante (à rapprocher de (3.2.3)) :

```
Derivee::Derivee(float x) : Base(20)
{
        .....
}
```

(5.2.7) Deuxième exemple

```
# include <iostream.h>

class Rectangle
{
public:
        Rectangle(short l, short h);
        void AfficheAire(void);
protected:
        short largeur, hauteur;
};

Rectangle::Rectangle(short l, short h)
{
        largeur = l;
        hauteur = h;
}
```

```
void Rectangle::AfficheAire()
{
        cout << "Aire = " << largeur * hauteur << "\n";
}

class Carre :  public Rectangle
{
public:
        Carre(short cote);
};

Carre::Carre(short cote) :  Rectangle(cote, cote)
{
}

void main()
{
        Carre *monCarre;
        Rectangle *monRectangle;
        monCarre = new Carre(10);
        monCarre -> AfficheAire();                    // affiche 100
        monRectangle = new Rectangle(10, 15);
        monRectangle -> AfficheAire();                // affiche 150
}
```

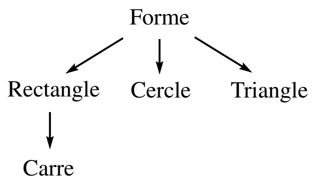
Nous retiendrons ceci :

Grâce à l'héritage, avec peu de lignes de code on peut créer de nouvelles classes à partir de classes existantes, sans avoir à modifier et recompiler ces dernières.

Le génie logiciel fait souvent usage de librairies toutes faites, contenant des classes qu'il suffit de dériver pour les adapter à ses propres besoins.

5.3 Polymorphisme

(5.3.1) Dans l'exemple précédent, `AfficheAire()` de `Carre` réutilise le code de `AfficheAire()` de `Rectangle`. Mais dans d'autres cas, il peut être nécessaire d'écrire un code différent. Par exemple, dans une hierarchie de classes de ce genre :



on a une version de `AfficheAire()` pour chacune de ces classes.

Si ensuite on crée une collection d'objets de type `Forme`, en demandant `AfficheAire()` pour chacune de ces formes, ce sera automatiquement la version correspondant à chaque forme qui sera appelée et exécutée : on dit que `AfficheAire()` est *polymorphe*. Ce choix de la version adéquate de `AfficheAire()` sera réalisé au moment de l'exécution.

Toute fonction-membre de la classe de base devant être surchargée (c'est-à-dire redéfinie) dans une classe dérivée doit être précédée du mot `virtual`.

(5.3.2) Exemple

```
# include <iostream.h>

class Forme
{
	// données et fonctions-membres....
public:
	virtual void QuiSuisJe(void);  // fonction destinée à être surchargée
};

void Forme::QuiSuisJe()
{
	cout << "Je ne sais pas quel type de forme je suis !\n";
}

class Rectangle :  public Forme
{
	// données et fonctions-membres....
public:
	void QuiSuisJe(void);
};

void Rectangle::QuiSuisJe()
{
	cout << "Je suis un rectangle !\n";
}

class Triangle :  public Forme
{
	// données et fonctions-membres....
public:
	void QuiSuisJe(void);
};

void Triangle::QuiSuisJe()
{
	cout << "Je suis un triangle !\n";
}

void main()
{
	Forme *s;
	char c;
	cout << "Voulez-vous créer  1 : un rectangle ?\n";
	cout << "                   2 : un triangle ?\n";
	cout << "                   3 : une forme quelconque ?\n";
	cin >> c;
	switch (c)
	{
		case '1' :  s = new Rectangle; break;
		case '2' :  s = new Triangle; break;
		case '3' :  s = new Forme;
	}
	s -> QuiSuisJe();      // (*) cet appel est polymorphe
}
```

Remarque.— Pour le compilateur, à l'instruction marquée `(*)`, il est impossible de savoir quelle version de `QuiSuisJe()` il faut appeler : cela dépend de la nature de la forme créée, donc le choix ne pourra être fait qu'au moment de l'exécution (on appelle cela *choix différé*, ou *late binding* en anglais).

(5.3.3) *Remarques :*

- Une fonction déclarée virtuelle doit être définie, même si elle ne comporte pas d'instruction.
- Un constructeur ne peut pas être virtuel. Un destructeur peut l'être.

(5.3.4) **Compatibilité de types**

Supposons déclaré :

```
class A
{
        .....
};

class B : public A      // ou private A
{
        .....
};

A a;
B b;
A *pa;
B *pb;
void *pv;
```

Alors :

- • l'affectation `a = b` est correcte ; elle convertit automatiquement `b` en un objet de type `A` et affecte le résultat à `a`
- l'affectation inverse `b = a` est illégale
- • de la même manière, l'affectation `pa = pb` est correcte
- l'affectation inverse `pb = pa` est illégale ; on peut cependant la forcer par l'*opérateur de conversion de type* `()`, en écrivant `pb = (B*) pa`
- • l'affectation `pv = pa` est correcte, comme d'ailleurs `pv = pb`
- l'affectation `pa = pv` est illégale, mais peut être forcée par `pa = (A*) pv`.

(5.3.5) **Opérateur de portée ::**

Lorsqu'une fonction-membre virtuelle `f` d'une classe `A` est surchargée dans une classe `B` dérivée de `A`, un objet `b` de la classe `B` peut faire appel aux deux versions de `f` : la version définie dans la classe `B` — elle s'écrit simplement `f` — ou la version définie dans la classe parente `A` — elle s'écrit alors `A::f`, où `::` est l'*opérateur de portée*.

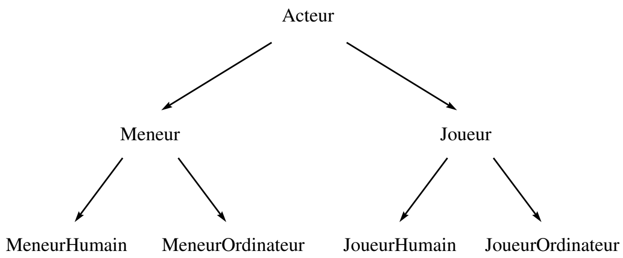
5.4 Exemple

(5.4.1) Reprenons le programme de jeu "c'est plus, c'est moins" du chapitre 3, §4. Nous pouvons maintenant écrire une version dans laquelle, au moment de l'exécution, les rôles du joueur et du meneur seront attribués soit à un humain, soit à l'ordinateur. L'ordinateur pourra donc jouer contre lui-même ! Il est important de noter que le déroulement de la partie, qui était contrôlé par la boucle :

```
do                              // voir la fonction main(), paragraphe (3.4.7)
{
        p = j.Propose(d);       // proposition du joueur
        d = m.Reponds(p);       // diagnostic du meneur
}
while (d && cpt < 6);
```

s'écrira de la même façon dans cette nouvelle version, quelle que soit l'attribution des rôles. Cela est possible grâce au polymorphisme des fonctions `Propose()` et `Reponds()`.

Nous utiliserons la hiérarchie de classes suivante :



(5.4.2) Conformément à (1.4.2), nous écrirons un fichier de déclaration `.h` et un fichier de définition `.cpp` pour chacune de ces sept classes. Afin d'éviter que, par le jeu des directives d'inclusion `#include`, certains fichiers de déclarations ne soient inclus plusieurs fois (ce qui provoquerait une erreur à la compilation), nous donnerons aux fichiers `.h` la structure suivante :

```
#ifndef SYMBOLE                         // si SYMBOLE n'est pas défini ...
#define SYMBOLE                         // ...  définir SYMBOLE
....  // ici, les déclarations normalement prévues
#endif                                  // fin du si
```

`#ifndef` est une *directive de compilation conditionnelle* : elle signifie que les lignes qui suivent, jusqu'au `#endif`, ne doivent être compilées que si `SYMBOLE` n'est pas déjà défini. Or, en vertu de la deuxième ligne, ceci n'arrive que lorsque le compilateur rencontre le fichier pour la première fois.

(5.4.3) **Déclarations des classes**

```
//---------------------- acteur.h ----------------------

#ifndef _ACTEUR_H
#define _ACTEUR_H

class Acteur
{
public:
        void AfficheNom();
protected:
        char nom[20];
};

#endif

//---------------------- meneur.h ----------------------

#ifndef _MENEUR_H
#define _MENEUR_H
```

```
#include "acteur.h"

class Meneur : public Acteur
{
public:
        virtual int Reponds(int prop);
};

#endif

//--------------------- joueur.h ---------------------

#ifndef _JOUEUR_H
#define _JOUEUR_H

#include "acteur.h"

class Joueur : public Acteur
{
public:
        virtual int Propose(int diag);
};

#endif

//--------------------- meneurhu.h ---------------------

#ifndef _MENEURHU_H
#define _MENEURHU_H

#include "meneur.h"

class MeneurHumain : public Meneur
{
public:
        MeneurHumain();
        int Reponds(int prop);
};

#endif

//--------------------- meneuror.h ---------------------

#ifndef _MENEUROR_H
#define _MENEUROR_H

#include "meneur.h"

class MeneurOrdinateur : public Meneur
{
public:
        MeneurOrdinateur();
        int Reponds(int prop);
private:
        int numsecret;
};

#endif

//--------------------- joueurhu.h ---------------------

#ifndef _JOUEURHU_H
#define _JOUEURHU_H
```

```
#include "joueur.h"

class JoueurHumain : public Joueur
{
public:
        JoueurHumain();
        int Propose(int diag);
};

#endif

//--------------------- joueuror.h ---------------------

#ifndef _JOUEUROR_H
#define _JOUEUROR_H

#include "joueur.h"

class JoueurOrdinateur : public Joueur
{
public:
        JoueurOrdinateur();
        int Propose(int diag);
private:
        int min, max, proposition;
};

#endif
```

(5.4.4) **Définitions des classes**

```
//--------------------- acteur.cpp ---------------------

#include "acteur.h"
#include <iostream.h>

void Acteur::AfficheNom()
{
        cout << nom;
}



//--------------------- meneur.cpp ---------------------

#include "meneur.h"

int Meneur::Reponds(int prop)
{
        // rien à ce niveau
}

//--------------------- joueur.cpp ---------------------

#include "joueur.h"

int Joueur::Propose(int diag)
{
        // rien à ce niveau
}

//--------------------- meneurhu.cpp ---------------------
```

```
#include "meneurhu.h"
#include <iostream.h>

MeneurHumain::MeneurHumain()
{
        cout << "Vous faites le meneur.  Quel est votre nom ?  ";
        cin >> nom;
        cout << "Choisissez un numéro secret entre 1 et 100.\n\n";
}

int MeneurHumain::Reponds(int prop)
{
        AfficheNom();
        cout << ", indiquez votre réponse :";
        int r;
        cout << "\n0 - C'est exact";
        cout << "\n1 - C'est plus";
        cout << "\n2 - C'est moins";
        cout << "\nVotre choix (0, 1 ou 2) ?  ";
        cin >> r;
        return r;
}

//--------------------- meneuror.cpp ---------------------

#include "meneuror.h"
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

MeneurOrdinateur::MeneurOrdinateur()
{
        strcpy(nom, "l'ordinateur");
        randomize();
        numsecret = 1 + random(100);
}

int MeneurOrdinateur::Reponds(int prop)
{
        cout << "Réponse de ";
        AfficheNom();
        if (prop < numsecret)
        {
                cout << " :  c'est plus\n";
                return 1;
        }
        else if (prop > numsecret)
        {
                cout << " :  c'est moins\n";
                return 2;
        }
        cout << " :  c'est exact\n";
        return 0;
}

//--------------------- joueurhu.cpp ---------------------

#include "joueurhu.h"
#include <iostream.h>

JoueurHumain::JoueurHumain()
{
        cout << "Vous faites le joueur.  Quel est votre nom ?  ";
```

```
        cin >> nom;
}

int JoueurHumain::Propose(int diag)
{
        AfficheNom();
        int p;
        cout << ", votre proposition ?  ";
        cin >> p;
        return p;
}

//--------------------- joueuror.cpp ---------------------

#include "meneuror.h"
#include <iostream.h>
#include <string.h>

JoueurOrdinateur::JoueurOrdinateur()
{
        strcpy(nom, "l'ordinateur");
        min = 1;
        max = 100;
        proposition = 0;
}

int JoueurOrdinateur::Propose(int diag)
{
        if (diag == 1)
                min = proposition + 1;
        else
                max = proposition - 1;
        proposition = (min + max) / 2;
        AfficheNom();
        cout << " propose :  " << proposition << "\n";
        return proposition;
}
```

(5.4.5) Traitement principal

```
//--------------------- jeu.cpp ---------------------

#include "meneur.h"
#include "meneurhu.h"
#include "meneuror.h"
#include "joueur.h"
#include "joueurhu.h"
#include "joueuror.h"
#include <iostream.h>
#include <conio.h>

void main()
{
        cout << "\n\n\tJEU DU C++, C--\n\n";
        Joueur *j;
        Meneur *m;

                        // distribution des rôles

        char rep;
        cout << "Qui est le meneur (h = humain, o = ordinateur) ?  ";
        cin >> rep;
        if (rep == 'h')
                m = new MeneurHumain;
```

```
        else
                m = new MeneurOrdinateur;
        cout << "Qui est le joueur (h = humain, o = ordinateur) ?  ";
        cin >> rep;
        if (rep == 'h')
                j = new JoueurHumain;
        else
                j = new JoueurOrdinateur;

                        // déroulement de la partie

        int p, d = 1, cpt = 0;
        do
        {
                p = j -> Propose(d);
                d = m -> Reponds(p);
                cpt++;
        }
        while (d && cpt < 6);

                        // affichage du résultat

        cout << "\nVainqueur :  ";
        if (d)
                m -> AfficheNom();
        else
                j -> AfficheNom();
        getch();
}
```

5.5 Héritage multiple

(5.5.1) En C++, il est possible de faire dériver une classe de plusieurs autres classes simultanément. On déclarera par exemple :

```
class A
{
        .....
};

class B
{
        .....
};

class C : public A, public B
{
        .....
};
```

La classe `C` hérite alors de `A` et `B` : une instance de la classe `C` possède *à la fois* les données et fonctions-membres de la classe `A` et celles de la classe `B`.

(5.5.2) Quand un objet de la classe `C` ci-dessus est créé, les constructeurs des classes parentes sont appelés : d'abord celui de `A`, ensuite celui de `B`. Quand un objet est détruit, les destructeurs des classes parentes sont appelés, d'abord celui de `B`, ensuite celui de `A`.

(5.5.3) Dans la situation ci-dessus, il peut arriver que des données ou fonctions-membres des classes `A` et `B` aient le même nom. Pour lever l'ambiguïté, on utilise l'opérateur de portée en écrivant par exemple `A :: x` pour désigner la donnée-membre `x` héritée de la classe `A`, et `B :: x` pour désigner celle qui est héritée de la classe `B`.

5.6 Classes abstraites

(5.6.1) Une fonction-membre virtuelle d'une classe est dite *purement virtuelle* lorsque sa déclaration est suivie de `= 0`, comme ci-dessous :

```
class A
{
        .....
        virtual truc machin() = 0;
        .....
};
```

Une fonction purement virtuelle n'a pas de définition dans la classe. Elle ne peut qu'être surchargée dans les classes dérivées.

(5.6.2) Une classe comportant au moins une fonction-membre purement virtuelle est appelée *classe abstraite*. A retenir :

Aucune instance d'une classe abstraite ne peut être créée.

L'intérêt d'une classe abstraite est uniquement de servir de "canevas" à ses classes dérivées, en déclarant l'interface minimale commune à tous ses descendants.

(5.6.3) **Exemple**

Il n'est pas rare qu'au cours d'un programme, on ait besoin de stocker temporairement des informations en mémoire, pour un traitement ultérieur. Une structure de stockage doit permettre deux actions principales :
- mettre un nouvel élément,
- extraire un élément.

On parle de *pile* lorsque l'élément extrait est le dernier en date à avoir été mis (structure *LIFO* pour *Last In, First Out*) et de *queue* ou *file d'attente* lorsque l'élément extrait est le premier en date à avoir été mis (structure *FIFO* pour *First In, First Out*).

Voulant programmer une classe `Pile` et une classe `Queue`, nous commencerons par écrire une classe abstraite `Boite` décrivant la partie commune à ces deux classes :

```
//---------------------- déclaration de la classe Boite ----------------------

class Boite             // classe abstraite décrivant une structure de stockage
                        // les éléments stockés sont de type "pointeur sur Objet"
                        // la classe Objet est supposée déjà déclarée
{
public:
        Boite(int n = 10);              // construit une boîte contenant au maximum n pointeurs
        ~Boite();                       // destructeur
        virtual void Mets(Objet *po) = 0;       // met dans la boîte le pointeur po
```

```
        virtual Objet *Extrais() = 0;  // extrait un pointeur de la boîte
        int Vide();                    // indique si la boîte est vide
        int Pleine();                  // indique si la boîte est pleine
protected:
        int vide, pleine;              // indicateurs
        int taille;                    // capacité de la boîte
        Objet **T;                     // considéré comme tableau de pointeurs sur Objet
};

//--------------------- définition de la classe Boite ---------------------

Boite::Boite(int n)
{
        taille = n;
        T = new Objet* [taille];       // on peut prévoir ici un test de débordement mémoire
        vide = 1;
        pleine = 0;
};

Boite::~Boite()
{
        delete [] T;                   // libère l'espace pointé par T
};

int Boite::Vide()
{
        return vide;
};

int Boite::Pleine()
{
        return pleine;
};

//--------------------- déclaration de la classe Pile ---------------------

class Pile :  public Boite             // classe décrivant une pile
{
public:
        Pile(int n = 10);              // construit une pile contenant au maximum n pointeurs
        void Mets(Objet *po);          // met dans la pile le pointeur po
        Objet *Extrais();              // extrait un pointeur de la pile
protected:
        int nbelements;                // nombre effectif d'éléments contenus dans la pile
};

//--------------------- définition de la classe Pile ---------------------

Pile::Pile(int n) :  Boite(n)
{
        nbelements = 0;
};

void Pile::Mets(Objet *po)
{
        T[nbelements++] = po;
        vide = 0;
        pleine = nbelements == taille;
}

Objet *Pile::Extrais()
{
        Objet *temp;
        temp = T[--nbelements];
        pleine = 0;
```

```
        vide = nbelements == 0;
        return temp;
}

//--------------------- déclaration de la classe Queue ---------------------

class Queue :  public Boite           // classe décrivant une queue
{
public:
        Queue(int n = 10);            // construit une queue contenant au maximum n pointeurs
        void Mets(Objet *po);         // met dans la queue le pointeur po
        Objet *Extrais();             // extrait un pointeur de la queue
protected:
        int tete,                     // indice où se trouve l'élément le plus ancien
            queue;                    // indice où se mettra le prochain élément
                                      // (T est utilisé comme un tableau circulaire)
};

//--------------------- définition de la classe Queue ---------------------

Queue::Queue(int n) :  Boite(n)
{
        tete = queue = 0;
};

void Queue::Mets(Objet *po)
{
        T[queue++] = po;
        queue %= taille;              // retour au début du tableau si nécessaire
        vide = 0;
        pleine = tete == queue;
}

Objet *Queue::Extrais()
{
        Objet *temp;
        temp = T[tete++];
        tete %= taille;               // idem
        pleine = 0;
        vide = tete == queue;
        return temp;
}
```

Voici par exemple comment nous pourrions utiliser la classe `Queue` :

- création d'une file d'attente contenant au plus 1000 éléments :

```
Queue *q = new Queue (1000);
```

- mise d'un élément dans la file :

```
if (! q -> Pleine())
        q -> Mets(pObjet);            // pObjet pointe sur un Objet supposé créé par ailleurs
```

- extraction d'un élément de la file :

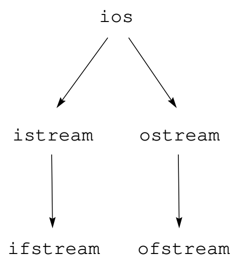
```
if (! q -> Vide())
        pObjet = q -> Extrais();
```

- destruction de la file :

```
delete q;
```

Chapitre 6

ENTREES & SORTIES



6.1 La librairie `iostream.h`

(6.1.1) **La classe `istream`**

En C++, un fichier est considéré comme un *flot* (en anglais : *stream*), c'est-à-dire une suite d'octets représentant des données de même type. Si ces octets représentent des caractères, on parle de *fichier-texte* ; si ces octets contiennent un codage en binaire, on parle de *fichier binaire*. Les organes logiques (clavier, console, écran) sont vus comme des fichiers-textes.

Les flots en entrée sont décrits par la classe `istream`.

L'objet `cin` est une instance de cette classe, automatiquement créé et destiné aux entrées depuis le clavier.

(6.1.2) En plus de l'opérateur de lecture `>>` que nous avons déjà utilisé, la classe `istream` dispose de nombreuses fonctions, dont les suivantes :

• `get()` 1$^{\text{ère}}$ forme, déclarée ainsi :

```
istream &get(char &destination);
```

C'est la lecture d'un caractère. La fonction renvoie une référence sur le flot en cours, ce qui permet d'enchaîner les lectures.

Exemple :

```
char c;
short nb;
cin.get(c) >> nb;      // si on tape 123 ⟨entrée⟩, c reçoit '1', nb reçoit 23
```

• `get()` 2$^{\text{ème}}$ forme, déclarée ainsi :

```
istream &get(char *tampon, int longueur, char delimiteur = '\n');
```

Lit au plus `longueur` caractères, jusqu'au délimiteur (inclus) et les loge en mémoire à l'adresse pointée par `tampon`. La chaîne lue est complétée par un `'\0'`. Le délimiteur n'y est pas inscrit, mais est remis dans le flot d'entrée.

Exemple :

```
char tampon[10];
cin.get(tampon, 10, '*');
```

• `get()` 3$^{\text{ème}}$ forme, déclarée ainsi :

```
int &get();
```

Lit un seul caractère, transtypé en `int`. On peut par exemple récupérer le caractère `EOF` (marque de fin de fichier) qui correspond à l'entier -1.

Exemple :

```
int c;
while ((c = cin.get()) != 'q')
       cout << (char) c;
```

• `getline()` déclarée ainsi :

```
istream &getline(char *tampon, int longueur, char delimiteur = '\n');
```

Lit une ligne. A la différence du `get()` 2$^{\text{ème}}$ forme, le délimiteur est absorbé au lieu d'être remis dans le flot d'entrée.

• `ignore()` déclarée ainsi :

```
istream &ignore(int longueur = 1, int delimiteur = EOF);
```

Elimine des caractères du flot d'entrée (fonctionne comme `getline()`).

Exemple :

```
char tampon[80];
cin.ignore(3).getline(tampon,80);
```

• `peek()` déclarée ainsi :

```
int peek();
```

Lit le caractère suivant sans l'enlever (fonctionne comme `get()` 3$^{\text{ème}}$ forme).

• `putback()` déclarée ainsi :

```
istream &putback(char c);
```

Remet le caractère désigné par `c` dans le flot (ce caractère doit être le dernier à avoir été lu).

• `seekg()` déclarée ainsi :

```
istream &seekg(streampos p);
```

Accès direct au caractère numéro `p`, ce qui permettra sa lecture ; les caractères sont numérotés à partir de 0. On peut préciser une position relative en mettant en second paramètre `ios::beg` , `ios::cur` ou `ios::end`.

• `read()` déclarée ainsi :

```
istream &read(void *donnees, int taille);
```

Lecture de `taille` octets depuis le flot, et stockage à l'adresse `donnees`.

• `gcount()` déclarée ainsi :

```
size_t gcount();
```

Renvoie le nombre d'octets lus avec succès avec `read()`.

(6.1.3) **La classe `ostream`**

Cette classe est destinée à décrire les flots en sortie.

L'objet `cout` est une instance de cette classe, automatiquement créé et destiné aux sorties à l'écran. `cerr` et `clog` en sont également deux instances, généralement associées à la console.

(6.1.4) En plus de l'opérateur d'écriture `<<` que nous avons déjà utilisé, la classe `ostream` dispose de nombreuses fonctions, dont les suivantes :

• `put()` déclarée ainsi :

```
ostream &put(char c);
```

Ecrit le caractère spécifié et renvoie une référence sur le flot en cours, ce qui permet d'enchaîner les écritures.
Exemple :

```
cout.put('C').put('+').put('+');      // affiche C++
```

• `seekp()` déclarée ainsi :

```
ostream &seekp(streampos p);
```

Accès direct à la position `p`, pour écriture. Comme pour `seekg()`, on peut mettre un second paramètre (voir (6.1.2)).

• `write()` déclarée ainsi :

```
ostream &write(const void *donnees, size_t taille);
```

Ecrit `taille` octets provenant de l'adresse `donnees`.

• `pgcount()` déclarée ainsi :

```
size_t pcount();
```

Renvoie le nombre d'octets écrits avec `write()`.

(6.2.5) **Utilitaires sur les caractères**

Voici quelques fonctions déclarées dans `<ctype.h>` concernant les caractères :

`tolower()`	convertit une lettre majuscule en minuscule
`toupper()`	convertit une lettre minuscule en majuscule
`isalpha()`	teste si un caractère est une lettre
`islower()`	teste si un caractère est une lettre minuscule
`isupper()`	teste si un caractère est une lettre majuscule
`isdigit()`	teste si un caractère est un chiffre entre 0 et 9
`isalnum()`	teste si un caractère est une lettre ou un chiffre.

6.2 La librairie `fstream.h`

(6.2.1) La classe `ifstream`

Cette classe décrit les fichiers en lecture. Elle dérive de `istream`, donc dispose des fonctions du paragraphe précédent, ainsi que des fonctions suivantes :

• un constructeur déclaré :

```
ifstream(const char *nom, int mode = ios::in);
```

Crée un nouvel objet de type `ifstream`, lui attache le fichier-disque appelé `nom` et ouvre ce fichier en lecture.

Exemple d'utilisation :

```
ifstream monfic("A:TOTO.TXT");
```

qu'on peut écrire de manière équivalente :

```
ifstream monfic;
monfic.open("A:TOTO.TXT");     // variante avec la fonction open()
```

Il est possible d'ouvrir le fichier en mode *ajout* (en anglais : *append*) pour pouvoir y ajouter des éléments à la fin. Il suffit pour cela de passer au constructeur comme second paramètre `ios::app`.

• `close()` qui ferme le fichier en fin de traitement.

(6.2.2) La classe `ofstream`

Cette classe décrit les fichiers en écriture. Elle dérive de `ostream`, donc dispose des fonctions du paragraphe précédent, ainsi que des fonctions suivantes :

• un constructeur déclaré :

```
ofstream(const char *nom, int mode = ios::out);
```

Crée un nouvel objet de type `ofstream`, lui attache un fichier-disque appelé `nom` et ouvre ce fichier en écriture.

• `close()` qui ferme le fichier en fin de traitement.

(6.2.3) **Exemple**

Voici une fonction qui recopie un fichier-texte.

```
void Copie(char *nomSource, char *nomDestination)
{
        ifstream source(nomSource);
        ofstream destination(nomDestination);
        char c;
        cout << "\nDébut de la copie...";
        while (source.get(c))          // explication en (6.3.2)
                destination << c;
        source.close();
        destination.close();
        cout << "\nCopie achevée.";
}
```

Remarquer qu'il ne faudrait pas lire les caractères de `source` par :

```
source >> c;
```

car l'opérateur `>>` sauterait les espaces et les marques de fin de ligne.

(6.2.4) **Remarques**

1. Pour créer un fichier binaire, il faut passer le mode `ios::binary` en second paramètre dans le constructeur (ou dans la fonction `open()`).

2. On peut combiner plusieurs modes d'ouverture avec l'opérateur `|`, par exemple `ios::in | ios::out`.

6.3 Fonctions de contrôle

(6.3.1) Pour tout flot, il est possible de contrôler le bon déroulement des opérations d'entrée-sortie, grâce aux fonctions suivantes :

• `good()` vraie si tout va bien et qu'en principe, la prochaine opération d'entrée-sortie devrait se dérouler normalement,

• `eof()` vraie si la dernière opération a fait atteindre la fin du fichier,

• `fail()` vraie s'il y a échec après une opération,

• `bad()` vraie s'il y a échec et si le fichier-disque est endommagé,

• `clear()` permettant de réinitialiser les bits d'état du flot.

(6.3.2) **Exemple 1**

La fonction `get()` renvoie en principe une référence de `stream` (voir (6.1.2)). Toutefois, dans le cas où une expression conditionnelle consiste en un appel à une fonction de `iostream` et lorsque cette fonction a pour valeur une référence de `stream`, le compilateur substitue à cette valeur le résultat de `good()`. Voici pourquoi la boucle de l'exemple (6.2.3) fonctionne correctement.

(6.3.3) **Exemple 2 : saisie protégée**

Voici un fragment de programme permettant de contrôler qu'une donnée introduite au clavier est correcte :

```
#include <iostream.h>
.....
short nombre;
cout << "Entrez un entier court :  ";
cin >> nombre;
if (cin.good())
        .....                  // traitement normal
else if (cin.fail())           // ce n'est pas une expression de type short
{
        cin.clear();           // on revient à l'état normal
        .....                  // message d'avertissement
}
```

6.4 Surcharge des opérateurs `>>` et `<<`

Le programme suivant montre comment on peut surcharger les opérateurs d'entrée-sortie habituels `>>` et `<<`, rendant ainsi possible la lecture (ou l'écriture) d'un objet depuis (ou vers) n'importe quel flot (fichier ou organe logique).

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>

const short MAX = 40;

class Plat
{
public:
        void Setnom(char *name);
        char *Getnom();
        void Setprix(float montant);
        float Getprix();
private:
        float prix;
        char nom[MAX];
};

void Plat::Setnom(char *name) { .....  }      // détails omis
char *Plat::Getnom() { .....  }
void Plat::Setprix(float montant) { .....  }
float Plat::Getprix() { .....  }

istream &operator>>(istream &is, Plat &article)
{
        float montant;
        char chaine[MAX];
        is.getline(chaine, MAX);       // mieux que is >> chaine
        article.Setnom(chaine);
        is >> montant;
        article.Setprix(montant);
        is.  ignore(1, '\n');
        return is;             // pour pouvoir enchaîner les entrées :  is >> a >> b ...
}
```

```
ostream &operator<<(ostream &os, Plat &article)
{
	os << article.Getnom() << " (F " << article.Getprix() << ")";
	return os;		// idem
}

void main()		// lit des plats depuis un fichier et les affiche à l'écran
{
	ifstream menu("MENU.TXT");
	Plat article;
	while (menu >> article)
		cout << article << "\n";
	menu.close();
}
```

6.5 Formatage des données

(6.5.1) Indicateurs

Les indicateurs suivants permettent de contrôler l'aspect des données en sortie :

`left`	alignement à gauche
`right`	alignement à droite
`fixed`	flottants en virgule fixe
`scientific`	flottants en notation scientifique
`showpoint`	force l'affichage avec virgule d'un flottant
`showpos`	force l'affichage d'un `+` devant un entier positif.

L'indicateur suivant permet de modifier le comportement de l'opérateur de lecture :

`skipws`	saute les blancs

Ces indicateurs de format sont membres de la classe `ios` et peuvent être réglés par les fonctions-membres `setf()` et `unsetf()` :

`setf(ios::`<*flag*>`)`	`active l'indicateur` <*flag*>
`unsetf(ios::`<*flag*>`)`	`désactive l'indicateur` <*flag*>

Exemple :

```
cin.setf(ios::skipws);		// saute les blancs en lecture au clavier
cin.unsetf(ios::skipws);	// ne saute pas les blancs en lecture au clavier
```

(6.5.2) Fonctions utiles

`width()`	détermine le nombre minimum de caractères de la prochaine sortie
`fill()`	précise le caractère de remplissage
`precision()`	détermine le nombre de chiffres après la virgule.

Exemple :

```
cout.width(12);
cout.fill('*');
cout.setf(ios::right);
cout << "Bonjour";		// affiche *****Bonjour
```

Autre exemple :

```
cout.width(8);
cout.precision(3);
cout << 100.0 / 3.0;           // affiche   33.333 (avec deux blancs devant)
```

(6.5.2) **Manipulateurs**

Ils permettent de modifier l'apparence des sorties et sont contenus dans la librairie `<iomanip.h>` :

`endl`	marque une fin de ligne et réinitialise le flot
`setfill(c)`	correspond à `fill()`
`setprecision(p)`	correspond à `precision()`
`setw(n)`	correspond à `width()`
`setbase(b)`	fixe la base de numération

Sauf pour `setprecision()`, ces manipulateurs n'agissent que sur la prochaine sortie.

Exemple :

```
cout << setbase(16) << 256 << endl;          // affiche 100 et passe à la ligne
cout << setprecision(2) << 123.45678;        // affiche 123.46
```

APPENDICE

A.1 Relation d'amitié

(A.1.1) On sait que lorsque des données et fonctions-membres d'une classe sont privées ou protégées, elles sont inaccessibles depuis l'extérieur de la classe : c'est le principe de l'encapsulation (cf. (1.3)). Si on a besoin d'y accéder, on peut écrire des fonctions réservées à cet effet (fonctions d'accès) et prévoir ainsi des contrôles et actions supplémentaires. Une autre possibilité, propre au C++, est d'utiliser une *déclaration d'amitié* avec le mot `friend`.

(A.1.2) **Fonctions amies**

Une *fonction amie d'une classe* est une fonction qui, sans être membre de la classe, a néanmoins accès à toutes les données et fonctions-membres de cette classe, qu'elles soient publiques, protégées ou privées.

Une telle fonction amie doit être déclarée à l'intérieur de la déclaration de la classe. S'il s'agit d'une fonction libre :

```
class A
{
        friend truc machin();  // fonction libre amie de la classe A
public:
                .....
private:
                .....
};
```

S'il s'agit d'une fonction-membre d'une autre classe :

```
class B;        // informe le compilateur qu'il existe une classe nommée B
class A
{
        friend truc B::machin();        // fonction-membre de la classe B, amie de la classe A
public:
                .....
private:
                .....
};
```

(A.1.3) Il peut être pratique d'utiliser une déclaration d'amitié pour surcharger certains opérateurs. Voici un exemple avec l'opérateur de sortie `<<` (voir (6.4)) :

```
class A
{
        friend ostream &operator<<(ostream &os, A &monObj);   // déclaration
private:
        int T[10];
        ......
};

ostream &operator<<(ostream &os, A &monObj)   // définition de la surcharge
{
        for (int i = 0; i < 10; i++)
                os << monObj.T[i];     // donnée accessible grâce à l'amitié
        return os;
}
```

(A.1.4) **Classes amies**

Lorsque toutes les fonctions-membres d'une classe `B` sont amies d'une classe `A`, on dit que la classe `B` est *amie de* `A`. Au lieu de déclarer dans la classe `A` chaque fonction-membre de `B` comme amie, on écrit plus simplement :

```
class B;        // informe le compilateur qu'il existe une classe nommée B
class A
{
        friend class B;               // la classe B est déclarée amie de la classe A
public:
                .....
private:
                .....
};
```

Remarque.— Il ne faut pas abuser de la relation d'amitié, car elle constitue une entorse au principe d'encapsulation.

A.2 Patrons

(A.2.1) En C++, il est possible de déclarer des classes paramétrées par des types, grâce au mécanisme des *patrons* (*template*). Supposons par exemple que nous voulions écrire une classe `Tableau` permettant de ranger aussi bien des entiers que des réels ou des chaînes de caractères. Au lieu d'écrire autant de classes `Tableau` qu'il y a de types à ranger, la solution consiste à écrire une unique classe `Tableau` paramétrée par un type a priori inconnu qu'on appelle `T` :

```
template <class T>     // signale que T est un type paramètre de ce qui suit
class Tableau
{
public:
        Tableau(short dim);
        ~Tableau();
        T &operator[](short index);            // surcharge de l'opérateur []
private:
        short taille;
        T *ptab;
};
```

Voici la définition de la classe `Tableau` :

```
template <class T>
Tableau<T>::Tableau(short dim)
{
        taille = dim;
        ptab = new T [taille];
};

template <class T>
Tableau<T>::~Tableau()
{
        delete [] ptab;         // libération-mémoire pour un tableau dynamique
};

template <class T>
T &Tableau<T>::operator[](short index)
{
        if (index < 0 || index > taille)
        {
                cout << "\nindice hors du rang...";
                exit(1);        // interrompt l'exécution
        }
        return ptab[index];
};
```

(A.2.2) Exemple d'utilisation de la classe `Tableau` précédente :

```
Tableau<int> t(10);     // déclare un tableau t contenant 10 entiers
int z;
z = t[1];               // ici, l'indice est automatiquement contrôlé
t[0] = 1;               // possible car la surcharge de [] est déclarée de type T&
```

Ou bien :

```
Tableau<float> u(3);    // déclare un tableau u contenant 3 réels
....
```

Ou encore :

```
typedef char Mot [20];
Tableau<Mot> t(100);    // déclare un tableau t contenant 100 mots
....
```

Remarques :

- le paramètre de type `T` peut être n'importe quel type : type de base, type défini ou classe
- chacune des déclarations précédentes provoque en réalité la recompilation de la classe `Tableau`, où le type paramètre `T` est remplacé par le type véritable
- une autre manière d'écrire une classe `Tableau` pouvant contenir différents types d'objets est d'utiliser l'héritage et le polymorphisme, comme nous l'avons fait pour les classes `Boite`, `Pile` et `Queue` au paragraphe (5.6.3).