# Le langage C++

## Méthodes numériques de la physique

T. Bastin & D. Baguette

# Table des matières

# 1 Préambule

Ce petit fascicule a pour objet de présenter l'essentiel de la syntaxe du langage C++. Cet exposé s'inscrit dans le cadre du cours de *Méthodes numériques de la physique* enseigné à la deuxième année des études de bachelier en sciences physiques. L'objectif du cours est l'acquisition et la mise en oeuvre de méthodes algorithmiques pour résoudre divers problèmes mathématiques rencontrés en physique. En ce sens, l'apprentissage et l'utilisation du C++ apparaît un peu comme un passage obligé tant ce langage se révèle être particulièrement bien adapté à l'implémentation de tels algorithmes. Le C++ produit en effet des programmes rapides et performants qui permettent d'exploiter au mieux toutes les potentialités de puissance offertes par un ordinateur (critère fondamental dans toute application sérieuse d'analyse numérique).

La programmation C++ peut être réalisée sous divers environnements (Windows, Linux, Mac OS, ...). Dans le cadre du présent cours, nous ne souhaitons pas exposer des techniques de programmation qui seraient liées à tel ou tel système d'exploitation. L'objectif est d'être capable de développer des programmes d'analyse numérique qui peuvent être compilés sous n'importe quel environnement. [1] C'est la raison pour laquelle cet exposé vise le seul apprentissage des techniques de programmation en environnement "ligne de commande" (programmes dits *consoles* qui ne comportent que des instructions "universelles" du C++). Toutes les techniques d'utilisation des fenêtres, boîtes de dialogue, etc. qui font appel à des instructions du langage spécifiques au système d'exploitation ne sont pas présentées et ne font pas partie de l'objectif du cours.

Nous avons fait le choix de ne pas produire un exposé exhaustif de toutes les possibilités offertes par le langage C++, car toutes ne sont pas utiles et indispensables à la création de programmes d'analyse numérique (même si l'essentiel y est présent). Nous renvoyons le lecteur à tout autre ouvrage spécialisé pour parfaire sa connaissance du langage et en maîtriser toutes ses subtilités.

1. afin notamment que les programmes développés dans le cadre des travaux pratiques puissent être ultérieurement réutilisés sous d'éventuels autres environnements.

# 2 Premier programme en C++

En langage C++, un programme consiste en l'exécution d'une suite d'instructions insérées dans le bloc principal

```
int main()
{
    return 0;
}
```

Ce bloc d'instructions doit être sauvegardé dans un fichier portant l'extension `.cpp`. Pour exécuter le programme, il faut tout d'abord le compiler afin de générer un fichier exécutable (la compilation est une opération consistant à transformer le texte du programme en code machine). Le programme est alors exécuté en lançant le fichier (`.exe` sous Windows, cliquer deux fois sur l'icône du fichier).

Pour tester le compilateur, écrivez le programme suivant qui doit afficher sur une fenêtre console "Ceci est mon premier programme en langage C++"

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Ceci est mon premier programme en langage C++";
    return 0;
}
```

Toutes les instructions se termine par un `;`. Les instructions peuvent être écrites sur plusieurs lignes. On peut écrire plusieurs instructions par ligne. Le nombre d'espaces entre les mots est sans importance.

Des commentaires peuvent être insérés dans un programme :

```
// Ceci est un commentaire sur une seule ligne

/*
Ceci est un commentaire
sur plusieurs lignes
*/
```

Les commentaires sont insérés par le programmeur afin de rendre le code-source plus clair ; ils sont ignorés par le compilateur. Les langages de programmation étant peu naturels, un programme bien commenté est plus facile à comprendre et à débugger.

# 3 Les types fondamentaux

Les données traitées en langage C++ sont stockées dans des variables. Selon la nature des données, ces variables sont de différents types.

## 3.1 Les types entiers

Les principaux types entiers sont `short`, `int` et `long`. On distingue aussi les types `unsigned short`, `unsigned int` et `unsigned long` qui ne présentent pas d'intérêt en analyse numérique.

| type | valeurs qui peuvent être stockées | occupation mémoire |
|---|---|---|
| `short` | entier entre $-32\,768$ et $32\,767$ | 2 octets |
| `long` | entier entre $-2\,147\,783\,648$ et $2\,147\,483\,647$ | 4 octets (ou 8 en 64 bits) |
| `int` | entier | 4 octets (ou 8 en 64 bits) |
| `unsigned short` | entier entre 0 et $65\,535$ | 2 octets |
| `unsigned long` | entier entre 0 et $4\,294\,967\,295$ | 4 octets (ou 8 en 64 bits) |
| `unsigned int` | entier $> 0$ | 4 octets (ou 8 en 64 bits) |

## 3.2 Les types flottants

Il n'y a que 2 types pour stocker les nombres réel (nombres en virgule flottante) : `float` et `double`. Seul le `double` est à utiliser pour les programmes d'analyse numérique.

| type | valeurs qui peuvent être stockées | chiffres significatifs |
|---|---|---|
| `float` | nombre ($>$ ou $<0$) entre $3.4 \cdot 10^{-38}$ et $3.4 \cdot 10^{38}$ | 6 |
| `double` | nombre ($>$ ou $<0$) entre $1.7 \cdot 10^{-308}$ et $1.7 \cdot 10^{308}$ | 14 |

## 3.3 Le type caractère

Il s'agit du type `char` qui permet de stocker un caractère. Il existe 256 caractères différents : toutes les lettres de l'alphabet (`'a'`, `'b'`, `'c'`, ..., `'A'`, `'B'`, `'C'`, ...), les caractères numériques (`'0'`, `'1'`, ...), les signes de ponctuation (`','`, `';'`, ...), en bref tous les caractères de la table ASCII. Il y a lieu de bien noter que les caractères représentés sont toujours placés entre apostrophes.

Il y a aussi les caractères dits *spéciaux*. Les plus importants sont le tabulateur (`'\t'`), le retour à la ligne (`'\n'`) et le caractère nul (`'\0'`) (à ne pas confondre avec le caractère zéro `'0'`). Il y a enfin 4 caractères avec une syntaxe particulière : les caractères `'\\'`, `'\''`, `'\"'` et `'\?'` qui représentent respectivement les signes \, ', " et ?. Le premier \ dans les séquences précédentes s'appelle le caractère d'échappement.

Le `char` est codé en mémoire sur 1 octet. Le code numérique utilisé pour stocker le `char` s'appelle le code ASCII. En voici quelques-uns :

| caractère | code ASCII | caractère | code ASCII |
|---|---|---|---|
| `'\0'` | 0 | `'0'`, `'1'`, ... | 48, 49, ... |
| `'\t'` | 9 | `'A'`, `'B'`, ... | 65, 66, ... |
| `'\n'` | 10 | `'a'`, `'b'`, ... | 97, 98, ... |

## 3.4 Le type booléen

Le type booléen `bool` permet de stocker les grandeurs logiques `true` et `false`. Les booléens sont stockés en mémoire sur 1 octet.

# 4 Les variables

## 4.1 La déclaration

Les données d'un programme sont stockées dans des variables (espace mémoire). Ces dernières doivent être déclarées (= donnée d'un nom et d'un type à la variable). Le nom d'une variable commence toujours par une lettre ou le caractère _ , puis contient n'importe quelle succession de lettres, de chiffres ou de caractères _ . Attention, les noms de variables sont sensibles à la casse : les majuscules et minuscules sont des lettres différentes. `ad1` et `aD1` sont des noms de variables différents.

Syntaxe de déclaration des variables : `type nom_var1 [, nom_var2, ...];`. Par exemple, on peut écrire dans le bloc `main()` :

```
int main() {
    short i;    // déclare 1 variable de type short nommée i
    int j;      //         1                   int          j
    int m, n;   //         2                   int          m et n
    double d;   //         1                   double       d
    char c;     //         1                   char         c
    bool b;     //         1                   bool         b

    return 0;
}
```

Attention, déclarer une variable ne fait que réserver l'emplacement en mémoire (en mémoire RAM) pour stocker une donnée. Contrairement à d'autres langages, la valeur de la variable est indéfinie après la déclaration (c'est-à-dire qu'elle peut valoir n'importe quoi et cela peut être une source de bugs dans un programme). Une variable numérique tout juste déclarée ne vaut pas nécessairement zéro. Il est donc important de prévoir l'initialisation de ses variables avant de les utiliser.

On initialise une variable soit dans la déclaration :

```
short i = 0;
double d = 1.;
char c = 'u';
int j, k = 1;       // attention, ici seul k est initialisé à 1
int j = 1, k = 1;   // ici j et k sont initialisés à 1
```

(noter bien le `.` après le 1 qui a toute son importance dans la déclaration de la variable `double` : `1.` désigne un nombre en virgule flottante (donc un type décimal), `1` désigne un entier), soit après la déclaration, par une opération d'affectation :

```
bool b;
b = true;
```

## 4.2 Les constantes

Une variable peut être déclarée constante en ajoutant le mot `const` avant le type de la variable :

```
const double PI = 3.14;
```

Dans ce cas, la variable ne peut plus être changée par la suite et il est donc obligatoire de lui donner une valeur dès la déclaration. Il devient interdit d'écrire par la suite une opération d'affectation :

```
const double PI = 3.14;
PI = 5.; // Erreur !! Le compilateur refusera de compiler le programme
```

Habituellement, les constantes sont représentées par des noms en majuscules.

## 4.3 Les pointeurs

Le contenu d'une variable est stocké en mémoire RAM en un certain emplacement. Chaque emplacement de cette mémoire possède une *adresse* bien déterminée. Cette adresse ne représente rien d'autre que le numéro d'octet où est stockée la variable. La mémoire RAM consiste en effet en une succession d'octets numérotés dont le nombre est déterminé par la taille de la mémoire (par exemple 256 "méga" de RAM signifie $256 \times 1024^2 = 256 \times 1\,048\,576 \simeq 256$ millions d'octets).

On obtient l'adresse d'une variable `a` par exemple en écrivant `&a`. Cette adresse peut être stockée dans une autre variable d'un type particulier : le type *pointeur*. Une variable de type pointeur reçoit comme contenu une adresse.

La déclaration d'une variable de type pointeur obéit à la syntaxe suivante. Si `p` veut recevoir par exemple l'adresse d'une variable de type `double`, on écrit

```
double *p;
```

Si `p` devait recevoir l'adresse d'une variable de type `int`, on écrirait

```
int *p;
```

Pour affecter au pointeur `p` l'adresse d'une variable `a` de type `double`, on écrira :

```
double a;
double *p;
p = &a;
```

On dit que `p` pointe vers `a`. Après l'instruction `p = &a;`, l'expression `*p` représente le contenu de `a`, c'est-à-dire le contenu de la variable dont l'adresse est stockée dans `p`. Les expressions

```
double a = 2.;
double c = a;
```

et

```
double a = 2.;
double c = *p;
```

produisent le même effet. Dans les 2 cas, le contenu de la variable `a` est stocké dans `c` qui vaut 2 après l'affectation. On peut aussi écrire l'instruction `*p = 3.;`, après quoi la variable `a` contient la valeur 3.

On peut définir plusieurs variables pointeurs qui contiennent l'adresse de `a` :

```
double a = 2.;
double *p1;
double *p2;
p1 = &a;
p2 = p1;    // OK parce que p1 et p2 pointent tous les 2 vers
            // un double
```

# 5 Les opérateurs

Un opérateur agit sur une ou deux variables (ou directement des données) et donne un résultat. Chaque opérateur n'agit que sur des données d'un certain type et donne un résultat également d'un certain type.

## 5.1 Les opérateurs arithmétiques

Les opérateurs arithmétiques sont les opérateurs `+`, `-`, `*`, `/` et `%` (modulo). Les 4 premiers agissent sur des types entiers ou flottants et donnent pour résultat un type entier ou flottant respectivement. Leur signification est explicite. Attention, si la division `/` agit sur 2 entiers, le résultat est l'entier qui représente la partie entière de la fraction.

L'opérateur `%` (modulo) agit sur des données de type entier et donne pour résultat le reste de la division entière.

Exemples d'utilisation de ces opérateurs :

```
double a, b = 2.;
int i = 1, j;

a = b * 5.;  // b est multiplié par 5. et le résultat (10.) est placé dans a
a = a / 4.;  // a est divisé par 4. et le résultat (2.5) est replacé dans a

j = i + 2;   // i est incrémenté de 2 et le résultat (3) est placé dans j
j = 5 / 2;   // j vaut 2 (division entière)
j = 5 % 2;   // j vaut 1 (reste de la division entière)
```

Il faut *toujours* travailler avec des données de *même* type. Au besoin, procéder à une conversion de type (voir ci plus loin) :

```
a = b / (double)j;
```

Si la conversion de type n'est pas écrite *explicitement*, le compilateur le fait *implicitement* (à votre place). La conversion implicite (par le compilateur) est souvent source de confusion et de bugs et n'est pas à recommander.

## 5.2 Les opérateurs d'affectation

Les opérateurs d'affectation sont les opérateurs `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `++`, `--`. Ils sont responsables de l'affectation d'une valeur à une variable.

L'opérateur d'affectation principal est l'opérateur `=` qui affecte directement une valeur à une variable (par exemple, au terme de l'instruction `i = 1;`, la variable `i` possède la valeur 1).

On peut aussi utiliser les opérateurs d'affectation combinés `+=`, `-=`, `*=`, `/=` et `%=`. La signification du premier opérateur est la suivante : `j += 2;` est équivalent à `j = j + 2;`. Au terme de cette instruction, le contenu de la variable `j` a été incrémenté de 2 et replacé dans la variable `j`. Les autres opérateurs fonctionnent de manière analogue.

Il y a aussi les opérateurs `++` et `--` qui signifient respectivement `+=1` et `-=1` sur des variables de type numérique. On écrit simplement `i++;` pour incrémenter `i` de 1 ou `i--;` pour le décrémenter de 1. On peut aussi écrire `++i;` et `--i;`.

## 5.3 Les opérateurs de comparaison

Les opérateurs de comparaison sont les opérateurs `>` (plus grand), `<` (plus petit), `>=` (plus grand ou égal), `<=` (plus petit ou égal), `==` (égal), `!=` (différent de). Ces opérateurs comparent des données de *même* type (2 nombres, 2 caractères, ...) et donne pour résultat une valeur booléenne (`true` ou `false`).

Nous pouvons par exemple écrire

```
bool b;
int i = 1;

b = (i > 0); // ici b sera true
```

## 5.4 Les opérateurs logiques

Les opérateurs logiques sont les opérateurs `&&` (et), `||` (ou) et `!` (not). Les 2 premiers agissent sur 2 booléens et rendent un booléen. Le dernier n'agit que sur 1 booléen et rend également un booléen. Voici un exemple de leur utilisation :

```
int i = 1;
bool b;
bool c = true;

b = (i > 0)&&(i < 1);   // ici b sera false
b = (i > 0)||(i < 1);   // ici b sera true
b = !c;                 // ici b sera false
b = !(i > 1);           // ici b sera true
```

## 5.5 Les conversions de type

Les opérateurs agissent sur des données de même type. Si on souhaite travailler avec des données de types différents, il y a lieu de convertir les types des données pour les rendre égaux. La conversion de type (ce que l'on appelle le *typecasting*) peut engendrer une perte de précision si on convertit une donnée vers un type moins précis.

Pour convertir une donnée d'un type donné vers un autre type, on écrit simplement le type souhaité entre parenthèses devant la donnée :

```
double a;
int i = 4, j;

a = (double)i * 2.4;
    // i est converti en double, puis multiplié par le flottant 2.4
    // a vaut 9.6 au terme de cette opération
j = (int)a;
    // a est converti en entier (j ne contient plus que la partie entière de a)
    // j vaut 9 au terme de cette opération
```

Voici en résumé les actions des conversions :

| | Conversions sans perte de précision |
|---|---|
| `short` | → `int`, `long` |
| `int` | → `long` |
| `short` | → `float` |
| `int`, `long` | → `double` |
| `float` | → `double` |

| | | Conversions avec perte de précision |
|---|---|---|
| `int`, `long` | → `short` | les nombres au-delà de 32767 sont perdus |
| `int`, `long` | → `float` | perte de chiffres significatifs si le nombre est trop grand |
| `float` | → `short` | seule la partie entière du `float` est conservée |
| `float`, `double` | → `int`, `long` | seule la partie entière du `float`, `double` est conservée |
| `double` | → `float` | perte de chiffres significatifs |

# 6 Les structures de contrôle

## 6.1 L'instruction `if`

L'instruction `if` exécute de manière conditionnelle une instruction ou un bloc d'instructions. La condition est déterminée par la valeur d'une expression booléenne (par exemple : `(i > 1)`).

Si une seule instruction dépend de la condition, on écrit :

```
if (condition)
    instruction;        // exécuté si condition est true
```

ou

```
if (condition)
    {
        instruction;    // exécuté si condition est true
    }
```

Si plusieurs instructions dépendent de la condition, on écrit :

```
if (condition)
    {
        instruction1;   // exécuté si condition est true
        instruction2;
        // ...
    }
```

## 6.2 L'instruction `if ... else`

L'instruction `if ... else` exécute de manière conditionnelle une instruction ou un bloc d'instructions. Si la condition n'est pas remplie une autre instruction ou bloc d'instructions est exécuté. La condition est une expression booléenne.

Si une seule instruction dépend de la condition, la syntaxe est la suivante :

```
if (condition)
    instruction; // ou {instruction;}   // exécuté si condition est true
else
    instruction; // ou {instruction;}   // exécuté si condition est false
```

Si plusieurs instructions dépendent de la condition, on écrit :

```
if (condition)
    {
    instructions;      // exécuté si condition est true
    }
else
    {
    instructions;      // exécuté si condition est false
    }
```

## 6.3 L'instruction `switch`

Illustrons sur un exemple le fonctionnement de l'instruction `switch` :

```
int i;

switch (i)
{
    case 0:
        instructions;  // instructions à exécuter si i = 0 (noter qu'il n'y a pas de { })
        break;         // Attention, important, bien écrire cette instruction
    case 1:
        instructions;  // instructions à exécuter si i = 1
        break;
    case 2:
    case 3:
    case 4:
        instructions;  // instructions à exécuter si i = 2, 3 ou 4
        break;
    default:
        instructions;  // instructions à exécuter si i est différent de 0, 1, 2, 3 et 4
        break;         // le bloc default n'est pas obligatoire
}
```

Les `case` doivent toujours être suivis de valeurs bien définies ou de constantes, mais jamais de variables.

## 6.4 L'instruction `while`

```
while (expression booléenne)
    instruction; // ou {instruction;} ou {instructions;}
```

signifie *tant que l'expression booléenne est vraie, exécuter l'instruction ou le bloc d'instructions.*

## 6.5 L'instruction `do ... while`

```
do
    instruction; // ou {instruction;} ou {instructions;}
while (expression booléenne);
```

signifie *exécuter l'instruction ou le bloc d'instructions, tant que l'expression booléenne est vraie.*

Par rapport au `while`, l'instruction ou le bloc d'instructions est exécuté au moins une fois avec le `do`.

## 6.6 L'instruction `for`

```
for (init_instruction; condition; reinit_instruction)
    instruction; // ou {instruction;} ou {instructions;}
```

L'instruction `for` fonctionne comme suit :

- l'instruction `init_instruction` est exécutée
- la condition (expression booléenne) `condition` est vérifiée
- si la condition est vraie, le bloc d'instructions est exécuté, sinon la boucle s'arrête
- l'instruction `reinit_instruction` est exécutée
- la condition `condition` est vérifiée
- si la condition est vraie, ...

Dans l'exemple suivant

```
for (i = 0; i < 10; i++)
    instruction; // ou {instruction;} ou {instructions;}
```

le bloc d'instructions est exécuté 10 fois. La première fois, `i` vaut `0`, puis `1`, ... jusque `9`.

Les boucles `while`, `do ... while` et `for` peuvent être interrompues prématurément avec l'instruction `break;` qui fait sortir directement le programme de la boucle. Cet usage doit néanmoins être restreint autant que possible.

# 7 Les tableaux

Un tableau est une zone continue en mémoire, constituée de cellules contenant des données toutes de même type.

## 7.1 Les tableaux unidimensionnels

### 7.1.1 Les tableaux statiques

Les tableaux unidimensionnels sont un moyen commode de stocker en mémoire un vecteur de $n$ composantes. Pour stocker 10 `double`, on utilise une variable tableau déclarée comme suit :

```
double a[10];
```

Le premier élément du tableau est `a[0]`, le second `a[1]`, ..., le dernier `a[9]`. Nous avons la représentation schématique suivante des 10 éléments du tableau `a` (le vecteur pourrait tout aussi bien être représenté verticalement) :

| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | a[8] | a[9] |
|---|---|---|---|---|---|---|---|---|---|

Les éléments d'une variable tableau sont indéterminés après la déclaration et contiennent au départ n'importe quelle valeur.

Pour initialiser les éléments du tableau, on écrit simplement

```
a[0] = 1.;
a[1] = 2.5;
// ...
```

On peut aussi initialiser les éléments d'un tableau directement lors de la déclaration :

```
double a[10] = {1., 25., 0., 2.4, 12., 3., 1., -4., -2., 6.};
```

En écrivant

```
double a[10] = {0.};
```

on initialise d'un coup tous les éléments du tableau à 0. Cette astuce ne fonctionne que pour la valeur 0.

Pour copier tous les éléments d'un tableau vers un autre tableau, il faut copier séparément chacun des éléments. Par exemple, si nous avons 2 tableaux `a` et `b` déclaré comme suit :

```
double a[10], b[10];
```

tous les éléments de `b` sont recopiés dans le tableau `a` de la façon suivante :

```
int i;
for (i = 0; i < 10; i++)
    a[i] = b[i];
```

Comme toute variable, un tableau est stocké en mémoire en un certain emplacement. L'adresse de la première cellule du tableau (c'est-à-dire l'adresse du début du tableau) est donnée par l'expression `&a[0]` ou tout simplement `a`. Cette adresse peut être stockée dans une variable de type pointeur et on peut par exemple écrire dans le cas du tableau `a` :

```
double *p;
p = &a[0];  // p contient l'adresse de la première cellule du tableau
p = a;      // idem (instruction équivalente à la précédente)
```

Lors de la déclaration d'un tableau, on est obligé d'indiquer dans le programme un nombre bien déterminé pour la dimension du tableau (ici `10`). On peut éventuellement utiliser une variable constante. Par exemple :

```
const int N = 10;
double a[N];
```

Par contre, on ne peut utiliser une variable non constante pour déclarer le nombre d'éléments d'un tableau. On ne peut écrire

```
int m;
m = 10;
double a[m]; // !! Erreur à la compilation
```

C'est la raison pour laquelle le tableau est dit *statique*. On ne peut créer un tableau statique avec un nombre d'éléments qui ne serait déterminé que pendant l'exécution du programme. Dans ce but, il faut créer un tableau dynamique.

### 7.1.2 Les tableaux dynamiques

Pour créer un tableau dynamique contenant 10 éléments de type `double`, on écrit

```
double *a = new double[10];
```

ou

```
double *a;
a = new double[10];     // cette instruction alloue dynamiquement
                        // (pendant l'exécution du programme)
                        // un espace mémoire pour le tableau.
                        // Cette instruction ne doit pas
                        // nécessairement suivre la déclaration double *a;
```

L'intérêt des tableaux dynamiques est qu'à présent on peut écrire

```
int m;
double *a;
...
m = 10;
a = new double[m];
```

On peut ainsi créer des tableaux dont la taille est déterminée par la valeur d'une variable calculée (ou dont la valeur est demandée à l'utilisateur) pendant l'exécution du programme.

On accède aux éléments d'un tableau dynamique de la même façon qu'un tableau statique. Le i-ème élément est initialisé à 5 par exemple en écrivant

```
a[i] = 5.;
```

On ne peut par contre initialiser les éléments d'un tableau dynamique directement lors de la déclaration de la variable comme c'est le cas pour les tableaux statiques.

**Particularité** Lorsqu'on a terminé d'utiliser une variable tableau dynamique, il *faut* (*toujours, toujours, toujours*) écrire l'instruction

```
delete[] a;
```

Cette instruction libère l'espace mémoire qui était occupé par le tableau dynamique.

Si on souhaite changer le nombre d'éléments d'un tableau dynamique, il *faut* d'abord faire disparaître le tableau au moyen de l'instruction `delete[]` (ce qui efface de facto tout son contenu), puis réallouer le nombre de cellules souhaité avec l'instruction `new`.

### 7.1.3 Les chaînes de caractères

En langage C, il n'existe pas de type particulier qui permette de stocker des données "chaîne de caractères", c'est-à-dire des données qui représentent des mots ou des phrases entières et non plus un seul caractère comme le type `char`.

Dans le langage C, on appelle chaîne de caractères tout tableau de caractères qui se termine par le caractère nul `'\0'`. Sans ce caractère nul, on se trouve en présence d'un simple tableau de caractères indépendants. Ainsi

```
char s[6] = {'c', 'o', 'u', 'c', 'o', 'u'};
```

ne représente pas une chaîne, mais simplement un tableau contenant les caractères 'c', 'o', 'u', 'c', 'o' et 'u'. Par contre,

```
char s[7] = {'c', 'o', 'u', 'c', 'o', 'u', '\0'};
```

est une chaîne qui représente le mot `"coucou"`.

Pour utiliser une chaîne de caractères en C++, il convient de définir un tableau de `char` dont le nombre d'éléments est au minimum égal au nombre de lettres du mot à stocker plus 1 (pour le caractère nul). Dans un tel tableau, le caractère nul marque la fin de la chaîne. Tout ce qui se trouve après est ignoré et n'est jamais pris en compte dans les opérations qui agissent sur les chaînes. Ainsi

```
char s[7] = {'b', 'a', 's', '\0', 'v', 'e', 'r'};
```

définit une chaîne contenant le mot `"bas"` et rien de plus. Si on souhaite afficher le contenu de la chaîne `s` avec l'instruction

```
cout << s;
```

c'est effectivement l'affichage de ce mot que l'on obtient.

On peut définir le contenu d'une chaîne dès la déclaration de manière moins lourde que la syntaxe précédente. Par exemple, pour définir une chaîne contenant le mot `"coucou"`, on peut simplement écrire

```
char s[7] = "coucou";   // coucou contient 6 lettres, mais il
                        // faut définir un tableau de 7 éléments
```

On ne peut par contre pas écrire

```
char s[7];
s = "coucou";   // Erreur !!
```

Le contenu de la chaîne doit être spécifié dès sa déclaration. Si l'on souhaite modifier son contenu par la suite, il faut utiliser les fonctions `strcpy` (copie de chaînes) et `strcat` (concaténation de chaînes) définies dans le fichier include `<cstring>` :

```
char s1[7], s2[7];
strcpy(s1, "cou");  // affecte "cou" à la chaîne s1
strcpy(s2, "cou");  // affecte "cou" à la chaîne s2
strcat(s1, s2);     // concatène la chaîne s2 à la chaîne s1
                    // s1 contient "coucou" au terme de l'opération
```

Quand on utilise les fonctions `strcpy` et `strcat`, il est important de leur passer comme paramètres des tableaux de caractères comportant un nombre d'éléments suffisants pour recueillir les chaînes souhaitées (ne jamais oublier la place pour le caractère nul).

La fonction `strcmp(s1,s2)` compare le contenu des chaînes `s1` et `s2` et rend pour résultat un nombre positif (si `s1` est lexicographiquement plus loin que `s2`), nul (si `s1` et `s2` sont identiques) ou négatif (si `s1` est lexicographiquement moins loin que `s2`).

La fonction `strlen(s)` donne la longueur de la chaîne `s`. La longueur de la chaîne est le nombre de caractères qu'elle contient, *caractère nul exclu*. Ainsi, au terme des instructions

```
char s[7] = "coucou";
int longueur = strlen(s);
```

la variable `longueur` contient 6.

Les différentes lettres d'une chaîne de caractères sont accessibles comme n'importe quel tableau. Ainsi `s[0]` est un `char` représentant la première lettre de la chaîne `s`, `s[1]` la seconde, et ainsi de suite. L'adresse en mémoire du tableau est donnée par `&s[0]` ou tout simplement par `s`.

## 7.2 Les tableaux bidimensionnels

Les tableaux bidimensionnels permettent de représenter des matrices. On déclare un tableau statique à 2 dimensions de 10 lignes et de 5 colonnes comme suit :

```
double a[10][5];
```

ou

```
const int M = 10;
const int N = 5;
double a[M][N];
```

Le premier indice représente la ligne, le second la colonne (comme pour les matrices). Le tableau `a` contient `M * N` éléments.

Pour affecter la valeur 5 à un élément `i`, `j` (i-ème ligne, j-ème colonne) du tableau, on écrit

```
a[i][j] = 5.;
```

On peut initialiser l'ensemble des éléments du tableau dès la déclaration. La matrice `a` suivante est initialisée à la matrice identité de dimension 3 :

```
double a[3][3] = {{1.,0.,0.},
                  {0.,1.,0.},
                  {0.,0.,1.}};
```

Dans cet exemple d'une matrice $3 \times 3$, le premier élément de la première ligne de la matrice (en haut à gauche) est l'élément `a[0][0]`, le dernier élément de la première ligne (en haut à droite) est l'élément `a[0][2]`, le premier élément de la dernière ligne (en bas à gauche) est l'élément `a[2][0]` et le dernier élément de la dernière ligne (en bas à droite) est l'élément `a[2][2]`. Nous avons la représentation schématique

| | | |
|---|---|---|
| a[0][0] | a[0][1] | a[0][2] |
| a[1][0] | a[1][1] | a[1][2] |
| a[2][0] | a[2][1] | a[2][2] |

En mémoire RAM, toutes les cellules d'un tableau bidimensionnel sont stockées côte à côte, *les lignes les unes après les autres*. Les 9 cellules du tableau précédent sont stockées comme suit :

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a[0][0] | a[0][1] | a[0][2] | a[1][0] | a[1][1] | a[1][2] | a[2][0] | a[2][1] | a[2][2] |

Par conséquent, on pourrait aussi représenter une matrice M x N via un tableau unidimensionnel :

```
const int M = 10;
const int N = 5;
double a[M * N];
```

Dans ce cas, l'élément `i`, `j` de la matrice est tout simplement donné par l'élément `a[i* N + j]` du tableau unidimensionnel. Si on veut affecter à cet élément la valeur 5, on écrit

```
a[i * N + j] = 5.;
```

Cette seconde façon de procéder moins intuitive est cependant l'unique façon pour créer un "tableau bidimensionnel" dynamique :

```
int m = 10;
int n = 5;

double *a;
a = new double[m * n];
```

L'élément `i`, `j` est initialisé à 5 en écrivant

```
a[i * n + j] = 5.;
```

Encore une fois, quand on n'a plus besoin du tableau dynamique, on écrit

```
delete[] a;
```

## 7.3 Les tableaux statiques multidimensionnels

On crée des tableaux à 3 dimensions ou plus suivant une technique analogue à celle utilisée pour les tableaux bidimensionnels.

```
double a[5][10][8];
```

crée un tableau à 3 dimensions contenant au total $5 * 10 * 8 = 400$ éléments.

## 7.4 Les vector

La gestion des tableaux dynamiques, telle que présentée à la section 7.1.2, requiert de ne pas oublier de supprimer la mémoire allouée pour le tableau. Par ailleurs, elle n'est pas optimale lorsqu'il s'agit de redimensionner un tableau existant. En effet, dans ce cas, il faut créer un nouveau tableau, y copier les éléments un par un depuis le tableau existant puis désallouer ce dernier.

La librairie standard du C++ fournit un type alternatif nommé vector pour gérer de façon plus aisée les tableaux de données de dimension variable. Les vector sont des tableaux dynamiques dont l'allocation et la libération de la mémoire est gérée automatiquement. De plus, ils disposent d'un certain nombre de fonctions qui rendent simples leur redimensionnement, copie, comparaison, accès et insertion d'éléments. La gestion de la mémoire est en outre adaptée à l'ordinateur sur lequel s'exécute le programme et permet un gain sensible de performances en cas de redimensionnement. Dans un projet nécessitant des tableaux dont la taille est initialement inconnue ou variable, on favorisera l'utilisation des vector aux tableaux dynamiques si cela est possible.

### 7.4.1 Déclaration d'objets de type vector

Afin d'utiliser les vector, il faut inclure la librairie standard du même nom dans l'en-tête du fichier source,

```
#include <vector>
```

Le type vector est défini dans l'espace de nom standard std, il faut donc s'assurer dans le fichier source de disposer de la ligne de code

```
using namespace std;
```

ou, si la ligne de code précédente n'est pas présente, préfixer tous les mots vector de std : :

Pour déclarer un vector nommé tab de 10 entiers (type int) initialement nul, on utilise la ligne de code suivante

```
vector<int> tab(10,0);  // déclare un vector de 10 éléments
                        // de type int initialement nuls
```

ou

```
std::vector<int> tab(10,0); // en l'absence de "using namespace std"
                            // dans le fichier source
```

Le type à stocker dans le vector est placé entre < et > juste après le mot vector. La déclaration est suivie du nom du tableau et de deux arguments optionnels entre parenthèses, respectivement, la taille du tableau et la valeur initiale de tous les éléments. Il est important de noter que, comme pour les tableaux dynamiques, la taille du tableau peut être la valeur d'une variable et peut donc être acquise ou calculée en cours d'exécution.

De façon générale, pour déclarer un vector nommé vec de $N$ éléments contenant des variables dont le type est T et dont la valeur initiale est $V$, on utilise la ligne de code

```
vector<T> vec(N,V);
```

On peut omettre le nombre d'éléments (le vector est alors de taille nulle et dit vide) et la valeur initiale (la valeur 0 est utilisée par défaut comme valeur initiale pour les nombres), mais en aucun cas on ne peut omettre le type des variables stockées dans le vector :

```
vector<int> myvec;   // est valide et déclare un tableau de taille nulle
vector<> myvec;      // n'est pas valide !
```

Il n'est pas nécessaire de supprimer la mémoire allouée pour un vector, celle-ci est automatiquement désallouée lorsque le vector n'est plus accessible dans le code. Ainsi, l'utilisation des vector ne nécessite jamais l'appel des opérateurs new et delete.

### 7.4.2 Accès aux éléments d'un vector

L'accès aux éléments d'un vector, en lecture et en écriture, peut se faire de la même façon que celle utilisée pour un tableau traditionnel, c'est-à-dire via l'opérateur [ ] où les indices du tableau commencent à 0. Le code suivant déclare un vector de 20 entiers et le remplit avec les nombres de 1 à 20.

```
vector<int> myvec(20); // déclaration d'un vector de 20 entiers
                       // initialement nuls

for(int i = 0; i < 20; i++)
    myvec[i]=i+1;
```

Comme pour les tableaux traditionnels, l'indice utilisé avec l'opérateur [ ] doit être strictement inférieur au nombre d'éléments contenus dans le vector. L'utilisation d'indices en dehors des bornes autorisées entraîne un résultat indéfini et mène dans la plupart des cas à une erreur !

Pour accéder à un élément, en lecture comme en écriture, on peut aussi utiliser la fonction membre at. Son utilisation est similaire à l'opérateur [ ] à ceci près qu'elle vérifie que l'indice passé soit dans les bornes d'allocation du tableau. Si ce n'est pas le cas, le programme s'arrête et génère une erreur du type "out of range" ce qui permet une détection précoce et efficace des erreurs.

```
vector<int> myvec(5); // déclaration d'un vector de 20 entiers
                      // initialement nuls

for(int i = 0; i < 5; i++)
    myvec.at(i)=i+1; // équivalent à myvec[i]=i+1;

myvec.at(6) ; // accès en dehors des bornes autorisée,
              // le programme s'arrête et affiche :
              // terminate called after throwing
```

```
// an instance of 'std::out_of_range'
// what():  vector::_M_range_check:
//__n (which is 6) >= this->size() (which is 5)
// Abandon
```

### 7.4.3 Taille et redimensionnement d'un vector

Les fonctions push_back et pop_back ajoute et supprime respectivement un élément à la fin du vector et redimensionnent automatiquement celui-ci de façon adaptée.

```
vector<int> myvec(5,0); // vector de taille 5
myvec.push_back(10);    // vector de taille 6 dont le dernier élément est 10
myvec.push_back(12);    // vector de taille 7 dont le dernier élément est 12
myvec.pop_back();       // vector de taille 6,
                        // le dernier élément a été supprimé
```

Il est aussi possible de demander explicitement de redimensionner (agrandir ou réduire) un vector en utilisant la fonction resize qui prend deux paramètres, la nouvelle taille du vector et la valeur par défaut des éléments ajoutés.

```
vector<int> myvec(5,1); // vector de 5 éléments égaux à 1
myvec.resize(10,2);     // vector de 10 éléments, les 5 premiers égaux à 1
                        // et les autres égaux à 2
myvec.resize(3);        // vector de 3 éléments  égaux à 1
```

La fonction size permet de déterminer la taille actuelle du vector, la fonction clear supprime tous les éléments du vector (après cette opération le vector est de taille nulle) et la fonction empty retourne true si le vector est vide, c'est-à-dire s'il est de taille nulle et ne contient donc aucun élément.

```
vector<int> myvec(5); // crée un vector de  5 éléments égaux à 0
myvec.push_back(1);   // ajoute un élément dont la valeur est 1
myvec.empty();        // teste si le vector est vide : ici retourne false
myvec.size();         // renvoie la taille du vector : ici 6
myvec.clear();        // supprime tous les éléments du vector
                      // et le redimensionne à une taille nulle
myvec.size();         // renvoie la taille du vector : ici 0
myvec.empty();        // teste si le vector est vide : ici retourne true
```

### 7.4.4 Copie et comparaison de vector

Les opérateurs de comparaison et de copie sont définis pour des vector *contenant des éléments de type identique* mais pas nécessairement de taille identique.

**Copie** L'opérateur = permet d'affecter un vector à un autre (contenant des éléments du même type) c'est-à-dire de rendre les deux vector identiques (en taille et en valeur des éléments)

```
vector<int> vec1(5,15); // vector de  5 éléments égaux à 15
vector<int> vec2;       // vector vide
vec2 = vec1;            // copie l'objet vec1 dans l'objet vec2
                        // ici vec2 contient maintenant 5 éléments égaux à 15
```

**Comparaison** L'opérateur == permet de comparer des vector contenant des éléments de type identique. L'opérateur de comparaison retourne true uniquement si les deux vector ont la même taille et que leurs éléments sont égaux deux à deux.

```
vector<int> vec1(5,1); // vector de  5 éléments égaux à 1
vector<int> vec2(6,1); // vector de  6 éléments égaux à 1
vector<int> vec3(5,3); // vector de  5 éléments égaux à 3
vec1==vec2; // compare les deux vector :
            // ici retourne false car leur taille est différente
vec1==vec3; // compare les deux vector :
            //ici retourne false car leurs éléments sont différents
vec2.pop_back(); // retire le dernier élément
vec1==vec2; // compare les deux vector : ici retourne true
            // car leur taille et leurs éléments sont identiques
```

### 7.4.5 Utiliser un vector comme un tableau 2D

Comme pour les tableaux dynamiques, il n'est pas possible de créer des vector à deux dimensions. On utilise donc la même technique que celle définie pour les tableaux dynamiques, c'est-à-dire qu'on alloue un vector à une dimension contenant le même nombre d'éléments que le tableau à deux dimensions et on utilise la correspondance suivante pour les indices : $k = i\,n + j$ où $k$ est l'indice à utiliser, $i$ l'indice de la ligne, $j$ l'indice de la colonnes et $n$ le nombre de colonnes.

```
int C = 5 ; // nombre de colonnes
int L = 10 ; // nombre de lignes

vector<int> mat(L*C) // représente une matrice L lignes et C colonnes
for(int i=0 ; i<L ; i++)
    for(int j=0 ; j<C ; j++)
        mat[i*C+j] = i+j ;
```

# 8 Les fonctions

## 8.1 La définition d'une fonction

Les instructions dans un programme C++ ne doivent pas toutes être incluses dans le bloc principal `main()` du programme. Des blocs séparés d'instructions peuvent être créés, blocs qui remplissent une tâche bien précise. Ces blocs peuvent ensuite être appelés à partir du bloc principal `main()`. Ils portent le nom de fonctions.

Illustrons d'emblée sur un exemple :

```
double plus(double x, double y)
{
    double res;
    res = x + y;
    return (res);
}

int main()
{
    double a, b, c;
    a = 5.;
    b = 10.;

    c = plus(a, b); // c contiendra 15.

    return 0;
}
```

Ce programme contient une fonction nommée `plus`. Le nom des fonctions obéit aux mêmes règles que les variables. Ici, la fonction `plus` admet 2 paramètres de type `double` : `x` et `y`. Elle retourne une donnée de type `double`.

On appelle un bloc d'instructions défini par une fonction simplement en écrivant son nom et en plaçant entre parenthèses les paramètres demandés. C'est réalisé ici par l'instruction `plus(a, b)` dans le programme principal. Lorsque le programme principal arrive à cette instruction, les opérations suivantes sont réalisées :

- le contenu de `a` est transféré dans `x` (ici 5.)
- le contenu de `b` est transféré dans `y` (ici 10.)
- les instructions dans le bloc de la fonction sont exécutées
- lorsque la fonction rencontre l'instruction `return`, le contrôle est rendu au programme principal.
- on place entre parenthèses dans l'instruction `return` le résultat de la fonction (ici un `double`)
- dans le programme principal, `plus(a, b)` contient alors la valeur donnée par le `return`
- cette valeur peut être placée dans une variable pour usage ultérieur (ici `c`).

Attention, `x` et `y` sont des copies *indépendantes* des variables `a` et `b`. Si la fonction modifie le contenu de `x` et `y`, `a` et `b` ne sont en rien affectés. Les variables sont dites *transmises par valeur*.

Il est possible de faire en sorte que les paramètres `x` et `y` ne soient pas des copies indépendantes de `a` et `b`, mais qu'au contraire ils les représentent directement de manière telle que toute modification de `x` et `y` dans la fonction se répercute sur les variables `a` et `b` en dehors de la fonction. On écrit dans ce cas `double plus(double &x, double &y)`. Les variables sont dites ici *transmises par adresse*.

Les variables déclarées dans une fonction (ici la variable `res`) ne peuvent être utilisées *que* dans le corps de la fonction. Elles n'ont aucune existence en dehors du corps de la fonction (ces variables sont dites *l*ocales). On dit qu'elles ont une *durée de vie* limitée à l'exécution de la fonction. Quand la fonction se termine, l'espace mémoire occupé par la variable disparaît. Une variable peut être utilisée partout (variable dite *g*lobale) si elle est déclarée en dehors de toute fonction (y compris en dehors de la fonction principale `main`). Dans l'exemple suivant, la variable constante `DIMENSION` est globale.

```
const int DIMENSION = 10;

double plus(double x, double y) {
    ...
}

int main() {
    ...
    return 0;
}
```

Une fonction ne contient pas nécessairement de paramètre. On écrit dans ce cas

```
double f1() {
    // instructions;
    return (2.);
}

int main()
{
    double c;
    c = f1(); // c contiendra 2.
    return 0;
}
```

Une fonction ne retourne pas nécessairement une valeur. On utilise dans ce cas le mot clé `void` :

```
void f2() {
    // instructions;
    return; // l'instruction return; est facultative
}

int main() {
    // instructions;
    f2(); // toutes les instructions situées dans le bloc f2 sont effectuées
    return 0;
}
```

L'instruction `return;` peut se trouver en plusieurs endroits dans la fonction. L'exécution de la fonction s'arrête dès qu'une de ces instructions est rencontrée. Pour une bonne lisibilité du code, il n'est pas toujours opportun d'utiliser cette possibilité.

Une fonction peut posséder comme paramètres des pointeurs. Une fonction peut être appelée dans une fonction. Une fonction peut s'appeler elle-même (*récursivité*). On peut définir autant de fonctions que l'on souhaite :

```
void f1() {
    // instructions;
    return;
}

double f2(double x) {
    // instructions;
    return (2. * x);
}

int main() {
    // instructions;
    return 0;
}
```

## 8.2 L'adresse d'une fonction

A l'instar de toute variable, le code d'exécution d'une fonction est stocké en mémoire en un emplacement bien précis. Pour désigner l'adresse de cet emplacement, on parle de l'*adresse de la fonction*. L'adresse d'une fonction est directement donnée par son nom. Par exemple, l'adresse de la fonction

```
bool f(double x, int n) {
    // instructions ...
}
```

est donnée par `f` (l'expression `&f` est également valable). Cette adresse peut être stockée dans une variable de type pointeur, appelée *pointeur de fonctions*. Dans le cas de la fonction `f`, pour qu'une variable de ce type (dénommée par exemple `fp`) puisse recevoir l'adresse de `f`, la déclaration doit s'écrire (attention, toutes les parenthèses ont leur importance)

```
bool (*fp)(double, int);
```

Suivant cette déclaration, `fp` désigne une variable pointeur pouvant recevoir l'adresse d'une fonction qui rend pour résultat un `bool` et qui prend comme paramètres d'abord un `double`, puis un `int`. Pour attribuer à `fp` l'adresse de la fonction `f` qui rentre dans cette catégorie, on écrit simplement

```
fp = f;
```

On peut par la suite invariablement écrire pour invoquer la fonction `f` et stocker son résultat dans une variable booléenne `b` :

```
b = f(3.14, 10);
```

ou

```
b = fp(3.14, 10);
```

Une fonction peut compter parmi ses paramètres l'adresse d'autres fonctions. Ceci permet en quelque sorte de transmettre à une fonction d'autres fonctions au même titre que n'importe quel paramètre transmis. Dans l'exemple suivant, la fonction `integrate` demande, outre les paramètres `a` et `b`, un paramètre `f` représentant l'adresse d'une fonction rendant un `double` et demandant comme paramètre un `double` :

```
double integrate(double (*f)(double), double a, double b) {
    // instructions
    c = f(a); // c reçoit le résultat de la fonction reçue en f
              // et évaluée en a
}
```

Si la fonction `integrate` a pour vocation de donner l'intégrale d'une fonction quelconque, on écrira par exemple pour intégrer la fonction cos(x) + sin(x) entre 0 et 1

```
double f1(double x) {
    return (cos(x) + sin(x));
}

int main() {
    double d;
    d = integrate(f1, 0, 1);
    return 0;
}
```

## 8.3 Des tableaux en tant que paramètres

### 8.3.1 Tableaux unidimensionnels

Une fonction peut posséder comme paramètre un tableau (statique ou dynamique). Pour transférer un tableau de `double`, la syntaxe de la fonction (ici nommée `f`) est donnée par

```
void f(double x[], int n) {
    // instructions;
    return;
}
```

où les `[]` indiquent que le paramètre `x` est un tableau unidimensionnel et l'entier `n` donne la taille du tableau que la fonction reçoit. L'appel de la fonction s'écrit comme suit :

```
int main() {
    double a[10];
    double *b = new double[100];
    // instructions;
    f(a, 10);       // le tableau statique a est transmis à la fonction,
                    // la fonction f sait que a possède 10 éléments
    f(b, 100);      // le tableau dynamique b est transmis à la fonction
                    //la fonction f sait que b possède 100 éléments
    delete[] b;
    return 0;
}
```

Attention, contrairement au cas de la transmission de variables de type simple, toutes les actions effectuées sur les éléments du tableau `x` dans la fonction se répercutent sur les éléments correspondants du tableau qui a été transmis (ici `a` et `b`). `x` *n'est pas* une copie du tableau transmis, il le représente *directement*. Une copie serait une opération beaucoup trop fastidieuse dans le cas de tableaux gigantesques. Ceci provient du fait que le paramètre transmis à la fonction est en réalité l'adresse du tableau et non le tableau lui-même (pour rappel `a` représente l'adresse de la première cellule, c'est-à-dire `&a[0]`).

Dans le cas où la fonction ne modifie pas les éléments du tableau, il est d'usage de faire précéder la déclaration du paramètre avec le mot clé `const` :

```
void f(const double x[], int n) {
    // instructions;
}
```

### 8.3.2 Tableaux multidimensionnels

Il n'est pas possible en langage C++ de transmettre des tableaux multidimensionnels dont la dimension est à priori quelconque. Si tel est le but à atteindre, il convient plutôt de travailler avec des tableaux unidimensionnels. Par exemple, pour transmettre à une fonction une matrice de dimension à priori quelconque $m \times n$, on représente cette matrice par un tableau unidimensionnel de dimension $m \times n$ et on utilise la technique de transmission des tableaux unidimensionnels, en transmettant toutefois comme paramètres à la fonction les 2 dimensions $m$ et $n$ afin que ces dernières soient connues à l'intérieur de la fonction. Par exemple, nous pouvons écrire :

```
void f(double x[], int m, int n) {
    // instructions;
    // x est un tableau unidimensionnel qui représente une matrice
    // à 2 dimensions m x n
    // Pour affecter 3 à l'élément i, j par exemple, on écrit
    ...
    x[i * n + j] = 3.;
    return;
}

int main() {
    int m = 3, n = 4;
    double *a = new double[m * n]; // déclaration d'un tableau unidimensionnel
                                   // dynamique pour représenter 1 matrice m x n
    // instructions;
    f(a, m, n);
    delete[] a;
    return 0;
}
```

### 8.3.3 L'objet vector comme paramètre et valeur de retour d'une fonction

Contrairement aux tableaux dynamiques, les vector se comportent de façon différente lorsqu'ils sont utilisés comme paramètres d'une fonction. Si on spécifie un vector comme paramètre d'une fonction, l'argument est copié dans la fonction et la modification du vector au sein de la fonction ne modifie pas le vector original. Comme la fonction size permet de connaître le nombre d'éléments contenu dans le vector, il n'est pas nécessaire d'ajouter un paramètre précisant le nombre d'élément à la fonction. Pour qu'un vector puisse être modifié par une fonction, il faut le passer par référence, c'est-à-dire rajouter le symbol "&" après le type déclaré dans la fonction.

```
void func1(vector<int> vec) // vec est une copie de l'argument
{
    for(int i=0; i<vec.size(); i++) // incrémente chaque élément
    {
```

```
        vec[i] += 1;
    }
}

void func2(vector<int>& vec) // vec est une référence vers l'argument
{
    for(int i=0; i<vec.size(); i++) // incrémente chaque élément
    {
        vec[i] += 1;
    }
}

int main()
{
    vector<int> v(10,5); // 5 éléments égaux à 5
    func1(v); // v n'est pas modifié par la fonction func1,
              // contient 5 éléments égaux à 6
    func2(v); // v est pas modifié par la fonction func2,
              // contient 5 éléments égaux à 6

    return 0 ;
}
```

Grâce aux vector, on peut utiliser un tableau comme valeur de retour d'une fonction. Ceci n'était pas recommandé dans le cas des tableaux dynamiques car il était ambigu de savoir quelle partie du code était chargée de libérer la mémoire. Ce problème n'existe plus avec les vector puisque la mémoire est libérée automatiquement. De plus les compilateurs C++ optimisent automatiquement les retours de vector comme paramètre de fonction afin de minimiser le nombre de copies.

```
// fonction qui retourne un vector dont la valeur des éléments est
// start, start+1, ..., end

vector<int> range(int start, int end)
{
    vector<int> myvec;
    for(int n = start; n<end ; n++)
    {
        myvec.push_back(n) ;
    }
    return myvec ;
}
```

## 8.4 La déclaration d'une fonction

Pour faire appel à une fonction, il est obligatoire que celle-ci soit définie *avant* l'appel de la fonction. Le programme suivant ne fonctionnera pas :

```
    int main()
    {
        double a, b, c;
        a = 5.;  b = 10.;
        c = plus(a, b);
```

```
    return 0;
}

double plus(double x, double y)
{
    double res;
    res = x + y;
    return (res);
}

// !! Erreur à la compilation
```

Il y a une astuce qui permet malgré tout de définir une fonction après son appel. L'astuce consiste à *déclarer* la fonction avant son premier appel en écrivant

```
double plus(double x, double y);  // déclaration de la fonction plus

int main()
{
    double a, b, c;
    a = 5.; b = 10.;
    c = plus(a, b);
    return 0;
}

double plus(double x, double y)   // définition de la fonction plus
{
    double res;
    res = x + y;
    return (res);
}
```

L'instruction `double plus(double x, double y);` déclare la fonction. Le compilateur sait que cette fonction sera définie plus loin et accepte alors de compiler. Noter bien le `;` dans la déclaration de la fonction.

Une fonction déclarée *doit* toujours être définie. Une fonction peut être déclarée autant de fois qu'on veut, mais ne peut être définie qu'une seule fois. Ainsi on peut écrire (c'est sans intérêt, mais c'est correct)

```
double plus(double x, double y);
double plus(double x, double y);
...
```

## 8.5 Les fichiers `.h` et `.cpp`

Quand on a beaucoup de fonctions à définir, il peut être intéressant de le faire dans un fichier séparé de celui contenant le bloc principal `main()`. Dans ce cas, il y a lieu de procéder comme suit :

– Les définitions de fonctions que l'on souhaite regrouper sont placées dans un fichier `.cpp` (par exemple : `numeric.cpp`)

- *Toutes* les déclarations de ces fonctions sont écrites dans un fichier dit d'*entête* `.h` portant usuellement le même nom que le fichier `.cpp` (ici `numeric.h`)
- Ces fichiers sont placés dans le même répertoire que celui du programme principal
- Pour faire appel à une quelconque de ces fonctions dans le programme principal, on écrit alors dans le cas de notre exemple (noter bien l'absence du `;` dans cette instruction)

```
#include "numeric.h"
```

Au cours de la compilation du programme, le compilateur remplace l'instruction `#include "numeric.h"` par l'ensemble du contenu du fichier `numeric.h`. Ce fichier contenant toutes les déclarations des fonctions de `numeric.cpp`, toutes ces fonctions deviennent appelables dans le bloc `main()`.

L'instruction `#include "numeric.h"` doit également être placée en tout début du fichier `numeric.cpp`.

## 8.6 Les bibliothèques de fonctions

Il est fourni en standard avec le compilateur de nombreux fichiers `.cpp` et `.h` contenant des définitions et déclarations de fonctions bien utiles. Par exemple, la librairie `cmath` contient toutes des déclarations de fonctions mathématiques :

- `cos(x)` (cosinus), `sin(x)` (sinus), `tan(x)` (tangente)
- `acos(x)` (arc cosinus), `asin(x)` (arc sinus(x)), `atan(x)` (arc tangente)
- `cosh(x)` (cosinus hyperbolique), `sinh(x)` (sinus hyperbolique), `tanh(x)` (tangente hyperbolique)
- `sqrt(x)` (racine carrée), `fabs(x)` (valeur absolue)
- `exp(x)` (exponentielle), `log(x)` (logarithme népérien : $ln$), `log10(x)` (logarithme en base 10 : $log$)

Toutes ces fonctions ont pour argument un `double` et rendent un `double`. Exemple d'utilisation :

```
double a;
double b = 1.;
a = 2. * cos(b);    // Attention, cos(b) calcule ici cosinus(1 radian)
```

Pour calculer le cosinus d'un angle en degrés, on écrit

```
const double pi = acos(-1.);    // pi est effectivement l'arc cosinus de -1
double angle = 45;
double a;
a = cos(angle * pi/180.);       // ici a représentera cosinus(45°)
```

Il y a aussi la fonction `pow(x,y)` qui calcule $x^y$ (`x` et `y` sont des `double` et le résultat est un `double`) et la fonction `abs(i)` qui donne la valeur absolue de l'entier `i`.

Pour faire appel à ces fonctions dans un fichier `.cpp` quelconque, on écrit en tête du fichier

```
#include <cmath>
```

avec la librairie à inclure (ici `cmath`) entre crochet (`< >`) et non plus entre guillemets (`" "`) comme dans le cas du fichier `numeric.h` de notre exemple ci plus haut. On utilise des crochets pour tous les fichiers fournis par le compilateur (ce qu'on appelle la librairie standard du compilateur) parce que ces fichiers ne se trouvent pas dans le même répertoire que celui de son programme `main()`. Dans le cas contraire (pour ses propres librairies), on utilise les guillemets.

# 9 Les classes

## 9.1 Notions de base

### 9.1.1 La déclaration et la définition d'une classe

Comme en java, on définit des classes en langage C++ (c'est l'utilisation des classes qui différencie le langage C du langage C++). L'utilisation et l'intérêt des classes en C++ sont en tout point similaires à ce qui se passe en java. L'idée est de pouvoir disposer d'objets pour lesquels diverses méthodes sont invoquées. En C++, la syntaxe d'une classe est la suivante (attention à la ponctuation qui a toute son importance) :

```
// définition de la classe
class figure {

    public:
        figure();       // cette méthode s'appelle le CONSTRUCTEUR de la classe
        ~figure();      // cette méthode s'appelle le DESTRUCTEUR de la classe

    public:
        double method1(double x);
        void method2();

    private:
        double method3();
        int m_k;            // donnée membre de la classe de type int
        double m_x;         // donnée membre de la classe de type double
        double *m_a;        // donnée membre pour définir un tableau dynamique
};


// définition de chacune des méthodes de la classe
figure::figure()
{
    instructions;
    // l'instruction return; NE PEUT PAS figurer dans ce bloc
}

figure::~figure()
{
    instructions;
    // l'instruction return; NE PEUT PAS figurer dans ce bloc
}

double figure::method1(double x) {
    ...
    return (...);
}

void figure::method2() {
    ...
    return;
}
```

```
double figure::method3() {
    ...
    return (...);
}
```

Il est d'usage de définir les classes dans des fichiers séparés du programme principal. La *définition de classe* est placée dans un fichier d'entête (par exemple `figure.h`), la *définition des méthodes* dans un fichier `.cpp` correspondant (dans l'exemple `figure.cpp`). On peut placer une ou plusieurs classes dans ces fichiers. Tout comme les fonctions, les classes peuvent être *déclarées* (autant de fois qu'on le souhaite). L'instruction suivante déclare la classe `figure` :

```
class figure;
```

Les méthodes des classes peuvent s'invoquer mutuellement. On peut par exemple écrire

```
double figure::method1(double x) {
    method2();
    ...
    return;
}
```

Il est également possible de définir les méthodes à l'endroit même de leur déclaration dans le fichier `.h`. La syntaxe suivante est correcte :

```
class figure {

    public:
        figure();       // cette méthode s'appelle le CONSTRUCTEUR de la classe
        ~figure();      // cette méthode s'appelle le DESTRUCTEUR de la classe

    public:
        double method1(double x) { ... }
        void method2();
    ...
};
```

Dans une telle déclaration, la `method1` est définie au niveau même de la déclaration et il n'est plus nécessaire de le faire dans le fichier `.cpp` correspondant. Habituellement, pour des questions de lisibilité du code, la définition des méthodes au niveau même de la déclaration de classe est réservée pour des méthodes ne comportant qu'un nombre extrêmement restreint d'instructions (1 ou 2 tout au plus).

En C++, on nomme généralement les données membres par un nom commençant par `m_`.

### 9.1.2 L'instance d'une classe (ou l'objet)

Une classe définit juste un modèle de variable (une sorte de type). Une variable de ce type s'appelle un *objet* (ou *instance de la classe*). Comme toute variable, il faut déclarer les objets. Le programme suivant déclare dans le module principal `main` un objet nommé `f` de la classe `figure` :

```
#include "figure.h"     // puisque la classe figure est déclarée
                        // dans ce fichier
```

```
int main() {
    figure f;
    ...
    return 0;
}
```

Une fois un objet déclaré, toutes les méthodes sous le mot clé `public` de la classe peuvent être invoquées. On invoque les méthodes avec l'opérateur point `.`. On peut par exemple écrire après la déclaration de `f`

```
double d = f.method1(1.);
```

mais pas

```
double ddd = f.method3();  // !! Erreur : method3() est private
```

car les méthodes déclarées dans la classe sous le mot clé `private` ne peuvent être invoquées par les objets de la classe. Les méthodes `private` sont seulement accessibles dans les méthodes de la classe (c'est-à-dire on ne peut les invoquer que là).

Les données membres d'une classe peuvent être des objets eux-mêmes.

Comme toute variable, l'objet `f` de l'exemple ci-dessus est stocké quelque part en mémoire RAM et l'adresse de son emplacement est donnée par `&f`. Cette adresse pourrait être stockée dans une variable `p` de type pointeur. On doit écrire dans ce cas :

```
figure f;
figure *p = &f;
```

On peut alors accéder aux méthodes qui s'appliquent à l'objet `f` à partir du pointeur `p`. On utilise dans ce cas l'opérateur flèche `->`. Les instructions suivantes produisent les mêmes effets :

```
f.method2();
p->method2();
```

Dans les 2 cas, la `method2` agit sur l'objet `f` parce que `p` contient l'adresse de `f`.

### 9.1.3 Le constructeur de classe

L'instruction `figure f;` dans le bloc `main()` ci-dessus déclare un objet `f` de type `figure`. Lorsque le programme rencontre une telle instruction de déclaration, le constructeur de la classe (la méthode `figure()`) est exécuté. C'est pourquoi les instructions que l'on écrit dans le constructeur sont habituellement des instructions d'initialisation des données membres de la classe (ici `m_k`, `m_x` et `m_a`). Si la classe possède comme donnée membre un tableau dynamique(`m_a` ici), il faut allouer l'espace mémoire pour le tableau dynamique. Dans notre exemple de la classe `figure`, nous pourrions écrire dans le constructeur

```
figure::figure() {
    int i;
    m_k = 0;                    // initialisation de m_k
    m_x = 0.;                   // initialisation de m_x
    m_a = new double[10];       // allocation de l'espace mémoire pour le
                                // tableau dynamique m_a
```

```
    for (i = 0; i < 10; i++)    // initialisation de tous les éléments
        m_a[i] = 0.;            // du tableau m_a
}
```

Le constructeur peut posséder un (ou plusieurs) paramètre(s). On peut par exemple avoir

```
class figure {
    figure(int n);
    ...
};
```

et définir le constructeur par

```
figure::figure(int n) {
    m_a = new double[n];
    ...
}
```

On déclare alors les objets suivant la syntaxe

```
int main() {
    figure f(5);    // appelle le constructeur de figure et transmet le paramètre 5
    figure g(10);   // appelle le constructeur de figure et transmet le paramètre 10
    ...
    return 0;
}
```

De cette façon, les objets `f` et `g` comportent chacun comme donnée membre un tableau dynamique `m_a` de 5 et 10 éléments respectivement.

S'il n'y a aucune instruction à placer dans le constructeur, il n'est obligatoire ni de le déclarer, ni de le définir.

### 9.1.4 Le destructeur de classe

À la fin du bloc `main()`, l'objet `f` qui, comme toute variable, occupait une certaine place mémoire, est effacé de la mémoire (pour rappel, les variables ont une durée de vie limitée). *Juste avant* la disparition de l'objet, le destructeur de classe (la méthode `~figure()`) s'exécute. On place habituellement dans le destructeur de classe des instructions de libération manuelle de mémoire. Par exemple, si la classe possède comme membre un tableau dynamique, on doit placer dans le destructeur l'instruction `delete` qui libère l'espace mémoire occupé par le tableau dynamique. En dehors de cette application, aucune autre instruction n'est utile dans le destructeur. Dans notre exemple de la classe `figure`, nous devons écrire

```
figure::~figure()
{
    delete[] m_a;
}
```

## 9.2 L'héritage

### 9.2.1 Les classes héritantes

On peut définir des classes (dites *classes héritantes*) qui dérivent d'autres classes (dites *classes parents*). Dans ce cas, la classe héritante possède toutes les méthodes et données

membres de la classe parent (cela signifie que tous les objets de la classe héritante peuvent invoquer les méthodes publiques de la classe parent), plus celles qu'elle se définit elle-même.

On peut par exemple déclarer une classe qui hérite de la classe `figure` comme suit :

```
class carre : public figure {
    public:
        void method4();
    private:
        int m_f;
};
```

La classe `carre` possède toutes les méthodes et données membres de la classe `figure`, plus 1 méthode `publique` (`method4`) et 1 donnée membre `private` (`m_f`) supplémentaires. La méthode supplémentaire doit bien entendu ensuite être définie dans un fichier `.cpp` :

```
void carre::method4() {
    ...
}
```

### 9.2.2 Le polymorphisme

La classe `carre` peut aussi *redéfinir* une méthode existant dans la classe `figure`. Supposons par exemple que l'on veuille redéfinir la `method2`, on écrit dans ce cas dans la classe parent `figure`

```
class figure {
    ...
    virtual void method2();     // le mot clé virtual doit juste apparaître ici
                                // dans la déclaration de la classe de base
                                // (et nulle part ailleurs)
};
```

et dans la classe héritante `carre`

```
class carre : public figure {
    ...
    void method2();
};
```

en n'oubliant pas de redéfinir la `method2` de la classe `carre` :

```
void carre::method2() {
    ...
}
```

L'intérêt d'une telle redéfinition réside dans le *polymorphisme*. Supposons que la `method1` de la classe parent `figure` invoque la `method2` :

```
double figure::method1(double x) {
    method2();
    ...
    return;
}
```

Soit alors un objet `c` de la classe héritante `carre` :

```
carre c;
```

Si l'objet `c` invoque la `method1` :

```
double d = c.method1(1.);
```

c'est bien entendu la `method1` définie dans la classe parent `figure` qui est exécutée puisque la classe `carre` ne redéfinit pas cette méthode. Au cours de l'exécution de cette méthode, la `method2` est invoquée. `c` étant de la classe `carre`, c'est la `method2` de cette classe qui est exécutée, bien que cette méthode soit appelée depuis la classe `figure`. Si par contre `c` avait été un objet de la classe `figure`, c'est la `method2` de la classe `figure` qui aurait été exécutée et non plus celle de la classe `carre` comme précédemment.

Bien que l'invocation de la `method2` n'apparaisse que dans la définition d'une méthode (`method1`) de la classe parent `figure`, la `method2` réellement exécutée dépend de l'objet qui invoque la `method1`. Cette dépendance constitue le *polymorphisme*. Cette propriété peut se révéler extrêmement intéressante dans de nombreuses applications.

### 9.2.3 Classes virtuelles pures

En principe, toutes les méthodes déclarées dans la définition d'une classe doivent être définies (dans le fichier `.cpp` ou au niveau même de la déclaration). Une situation particulière existe où cela n'est pas le cas.

En écrivant

```
class figure {
    ...
    virtual void method2() = 0;
};
```

la méthode `method2` de la classe `figure` ne peut plus être définie nulle part. La classe `figure` devient une *classe virtuelle pure*. Cela signifie qu'aucun objet de cette classe ne peut être déclaré. Seuls peuvent l'être des objets de classes dérivées pour autant que les classes dérivées définissent correctement la (ou les) méthode(s) restée(s) indéfinie(s) dans la classe parent (ici `method2`).

# 10 Les entrées-sorties standards (écran, clavier et fichiers)

Un programme qui n'interagit pas du tout avec l'extérieur ne peut présenter aucun intérêt. Il faut au minimum que le programme puisse écrire un résultat à l'écran ou dans un fichier. Dans ce but, la technique des flux est utilisée.

## 10.1 Écrire à l'écran (sur la console)

### 10.1.1 Le flux `cout`

Pour écrire le contenu de n'importe quelle variable à l'écran, on utilise l'objet `cout` qui est une instance de la classe `ostream_withassign` définie dans la libraire de flux d'entrée-sortie `iostream` (`io` signifie input-output, `stream` signifie flux). Le flux `cout` doit être vu comme un "tube" (un "pipe" en anglais) qui relie le programme avec l'écran.

Pour afficher le contenu d'une variable à l'écran, on procède comme suit :

- placer le contenu de la variable dans le tube `cout` (avec l'opérateur `<<`)
- vider le tube et envoyer son contenu vers l'écran (avec l'invocation de la méthode `flush()` de l'objet `cout`)

Par exemple, voici un programme qui imprime à l'écran le contenu d'une variable `double` (remarquez que `iostream` s'écrit sans le `.h` et que l'utilisation du flux `cout` nécessite l'instruction préalable `using namespace std;` :

```
#include <iostream>

using namespace std;

int main() {
    double d = 2.;
    cout << d;      // place le contenu de d dans le "tube" cout
    cout.flush();   // vide le tube et envoie son contenu à l'écran
    return 0;
}
```

On peut aussi écrire `cout << flush;` à la place de `cout.flush();`

On peut placer simultanément plusieurs données dans le tube :

```
double d = 2.;
cout << d << 5 << 2.*d << "coucou" << 't';
```

ou successivement :

```
double d = 2.;
cout << d;
cout << 5;
cout << 2.*d;
cout << "coucou";
cout << 't';
```

Le résultat est identique. Après une instruction `cout.flush();`, toutes ces données (dans le tube) sont affichées à l'écran l'une derrière l'autre : `254coucout`.

On peut placer un tabulateur entre chaque donnée en insérant le caractère `'\t'` :

```
cout << 2. << '\t' << 5;
cout.flush();
```

donne à l'écran `2 5`.

On peut aussi insérer un retour à la ligne `'\n'` sur les données affichées.

```
cout << 2. << '\t' << 5 << '\n';
cout << 3. << '\t' << 6;
cout.flush();
```

donne

```
2   5
3   6
```

Pour afficher une matrice 3 x 3, on écrit

```
double a[3][3];
int i, j;
for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
        cout << a[i][j] << '\t';
    }
    cout << '\n';
}
cout.flush();
```

Si les éléments de `a` sont tous nuls par exemple, on obtient à l'écran

```
0   0   0
0   0   0
0   0   0
```

### 10.1.2 Formater l'affichage des nombres à virgules flottantes

Il est possible d'agir sur l'affichage des nombres réels en utilisant des "manipulateurs" définis dans la librairie `iomanip`.

On choisit le nombre $n$ de chiffres significatifs à afficher en plaçant dans le flux de sortie le manipulateur `setprecision(n)`. Par défaut, $n = 6$.

Par exemple,

```
#include <iostream>         // pour cout
#include <cmath>            // pour sqrt

using namespace std;

int main() {
    cout << sqrt(2.) << flush;
    return 0;
}
```

affiche `1.41421`

```
#include <iomanip>        // pour setprecision
...
    cout << setprecision(2) << sqrt(2.) << flush;
```

affiche `1.4`

```
cout << setprecision(12) << sqrt(2.) << flush;
```

affiche `1.41421356237`

On peut forcer l'affichage scientifique en utilisant le manipulateur `setiosflags(0x800)`. A l'inverse, `setiosflags(0x1000)` empêche l'affichage scientifique en toute circonstance.

```
cout << setiosflags(0x800) << sqrt(2.) << flush;
```

affiche `1.414214e+000`

On peut forcer l'affichage de tous les chiffres significatifs, même s'ils sont nuls, par le manipulateur `setiosflags(0x100)`.

```
cout << 2. << flush;
```

affiche 2, tandis que

```
cout << setiosflags(0x100) << 2. << flush;
```

affiche `2.00000`.

Pour provoquer un affichage des nombres en colonne, il est intéressant d'utiliser le manipulateur `setw(n)` où $n$ spécifie l'espace réservé pour l'affichage de chaque nombre. Dans ce cas, tous les nombres sont collés à la droite d'une colonne fictive de $n$ caractères. Le manipulateur `setw(n)` doit être indiqué devant *chaque* nombre à afficher.

```
cout << 2. << '\t' << 3. << '\n';
cout << -2.4255 << '\t' << -3.54 << flush;
```

affiche

```
2       3
-2.4255 -3.54
```

tandis que

```
cout << setw(12) << 2. << setw(12) << 3. << '\n';
cout << setw(12) << -2.4 << setw(12) << -3.5 << flush;
```

donne le résultat plus lisible (surtout lorsqu'il y a beaucoup de nombres)

```
           2           3
     -2.4255       -3.54
```

### 10.1.3 Faire une pause en attendant une entrée clavier

Selon le système d'exploitation (dont Windows), à la fin du programme, la fenêtre console disparaît instantanément et ne laisse pas le temps à l'utilisateur de voir les résultats affichés par le flux `cout`. Pour éviter cet inconvénient, on écrit tout à la fin du programme

```
#include <iostream>

using namespace std;

int main() {
    ...
    cout << "\n\nPress a key to continue" << endl;
    cin.ignore();
    cin.get();
    return 0;
}
```

Ces instructions font appel au flux `cin` qui est une instance de la classe `istream_withassign` définie dans le fichier `iostream`. Ce flux nécessite préalablement l'instruction `using namespace std;`. La fonction `cin.get()` attend une entrée de la part de l'utilisateur. Tant que celui-ci n'appuye pas sur une touche, le programme reste en attente. La fonction `cin.ignore()` permet de vider préalablement le flux `cin` au cas où il aurait été utilisé auparavant.

## 10.2 Introduire des données dans le programme depuis le clavier

Pour introduire des données depuis le clavier, on utilise le flux `cin` qui est une instance de la classe `istream_withassign` définie dans le fichier `iostream`. Pour introduire une donnée réelle par exemple, on procède comme suit :

- on déclare une variable `double` (par exemple : `double x;`)
- on écrit l'instruction `cin >> x;`

L'instruction `cin >> x;` attend que l'utilisateur tape une donnée au clavier. Les caractères introduits sont affichés à l'écran. La saisie de la donnée est terminée lorsque l'utilisateur tape sur la touche `Enter` et l'affichage à l'écran passe à la ligne. Dès ce moment, la donnée est envoyée dans le tube `cin` et "aboutit" dans la variable `x`.

On écrira par exemple pour introduire une donnée au clavier (l'utilisation du flux `cin` nécessite également l'instruction préalable `using namespace std;` :

```
#include <iostream>

using namespace std;

int main() {
    double x;
    cout << "Veuillez introduire un nombre réel : " << flush;
    cin >> x;
    return 0;
}
```

Au terme de ces instructions, la variable `x` contient le nombre réel introduit au clavier.

Pour introduire un tableau de 3 nombres réels, on peut écrire

```
    double x[3];
    int i;
    for (i = 0; i < 3; i++) {
        cout << "Veuillez introduire x[" << i << "] : " << flush;
        cin >> x[i];
    }
```

## 10.3 Écrire dans un fichier

L'écriture de données dans un fichier est une opération extrêmement simple, analogue à l'écriture de données à l'écran. Plutôt que d'utiliser le flux `cout` qui aboutit à l'écran, on utilise un flux qui aboutit dans le fichier désiré.

Concrètement, on déclare et on utilise un flux de sortie de la classe `ofstream` définie dans le fichier `fstream` (`f` pour file, et `stream` pour flux). L'opérateur de transfert de données dans le flux de sortie est l'opérateur `<<`. Cet opérateur est utilisé exactement comme dans le cas du flux `cout`.

Voici la suite d'instructions (4 au total) pour écrire un nombre dans le fichier `test.txt` :

```
#include <fstream>      // pour utiliser les flux vers des fichiers

int main() {
    ofstream f;         // définit un flux (un "tube") f vers un fichier

    f.open("test.txt"); // crée le fichier "test.txt" s'il n'existe pas
                        // ou ouvre ce fichier s'il existe (et efface alors
                        // d'emblée tout son contenu antérieur).
                        // Au terme de l'instruction, le tube f est dirigé
                        // vers le fichier "test.txt".

    f << 2.;            // Ecrit 2. dans le fichier
    // ...              // Ecrire tout ce qu'on souhaite comme avec cout

    f.close();          // clôture des opérations et fermeture du fichier
                        // Il ne reste plus qu'à aller voir ce qu'il y a dedans!
    return 0;
}
```

Pour écrire dans un fichier, l'instruction `flush` n'est pas nécessaire. Le tube se vide automatiquement lors de la fermeture du fichier (à l'inverse de ce qui se passe pour `cout`). Le fichier `test.txt` est créé dans le répertoire de travail, généralement le répertoire contenant le programme. Si on souhaite un autre répertoire, il faut le mentionner explicitement. Par exemple

```
    f.open("C:\\rep\\test.txt"); // sous Windows
    f.open("/home/user/rep/test.txt"; // sous Unix
```

crée un fichier `test.txt` dans le répertoire `rep`. Pour rappel, le caractère anti-slash \ est représenté par le caractère `'\\'`.

## 10.4 Lire les données d'un fichier

Pour lire les données contenues dans un fichier, il faut déclarer et utiliser un flux d'entrée de la classe `ifstream` définie dans le fichier `fstream`. Comme `cin`, l'opérateur de transfert

de données dans le flux d'entrée est l'opérateur `>>`.

Supposons que le contenu du fichier `test.txt` soit

```
2   3.2
4   5.2
```

on obtient ces 4 valeurs dans un programme comme suit :

```
#include <fstream>          // pour utiliser les flux depuis des fichiers

void main() {
    int i;
    double x[4];            // définit un tableau de 4 doubles pour accueillir
                            // les 4 nombres du fichier
    ifstream f;             // définit un flux (un "tube") depuis un fichier

    f.open("test.txt");     // ouvre le fichier "test.txt" et fait pointer
                            // le flux f vers lui. Si le fichier n'existe pas
                            // il est créé.

    for (i = 0; i < 4; i++) // lit successivement les 4 valeurs et les place
        f >> x[i];          // dans le tableau x. Les espaces entre les nombres
                            // ne sont jamais considérés.

    f.close();              // clôture des opérations et fermeture du fichier
}
```