C++ : LES BASES

SOMMAIRE :

Chapitre 1

PRESENTATION DU C++

Apparu au début des années 90, le langage C++ est actuellement l'un des plus utilisés dans le monde, aussi bien pour les applications scientifiques que pour le développement des logiciels. En tant qu'héritier du langage C, le C++ est d'une grande efficacité. Mais il a en plus des fonctionnalités puissantes, comme par exemple la notion de classe, qui permet d'appliquer les techniques de la programmation-objet.

Le but de ce cours est de présenter la syntaxe de base du langage C++. Certains traits propres au C, dont l'usage s'avère périlleux, sont passés sous silence. La programmation-objet, quant à elle, sera abordée dans un autre cours.

1.1 Qu'est-ce qu'un programme ?

(1.1.1) 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*, puis, avant d'être exécutées, sont traduites en *langage machine* (qui est le langage du microprocesseur). Cette traduction s'appelle *compilation* et elle est effectuée automatiquement par un programme appelé *compilateur*.

Pour le programmeur, cette traduction automatique implique certaines contraintes :
- il doit écrire les instructions selon une syntaxe rigoureuse,
- il doit déclarer les données et fonctions qu'il va utiliser (ainsi le compilateur pourra réserver aux données une zone adéquate en mémoire et pourra vérifier que les fonctions sont correctement employées).

(1.1.2) Un programme écrit en C++ se compose généralement de plusieurs *fichiers-sources*. Il y a deux sortes de fichiers-sources :
- ceux qui contiennent effectivement des instructions ; leur nom possède l'extension `.cpp`,
- ceux qui ne contiennent que des déclarations ; leur nom possède l'extension `.h` (signifiant "header" ou *en-tête*).

(1.1.3) Un fichier `.h` sert à regrouper des déclarations qui sont communes à plusieurs fichiers `.cpp`, et permet une compilation correcte de ceux-ci. Pour ce faire, dans un fichier `.cpp` on prévoit l'inclusion automatique des fichiers `.h` qui lui sont nécessaires, grâce aux *directives de compilation* `#include`. En supposant que le fichier à inclure s'appelle `untel.h`, on écrira `#include <untel.h>` s'il s'agit d'un fichier de la bibliothèque standard du C++, ou `#include "untel.h"` s'il s'agit d'un fichier écrit par nous-mêmes.

(1.1.4) Le lecteur peut dès à présent se reporter au § 2.5 (page 8) pour voir un exemple de programme écrit en C++. Il convient de noter que :

- contrairement au Pascal, *le C++ fait la différence entre lettres minuscules et majuscules* : par exemple, les mots `toto` et `Toto` peuvent représenter deux variables différentes.
- 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,
- tout programme comporte une fonction (et une seule) appelée `main()` : c'est par elle que commencera l'exécution.

1.2 Environnement de développement intégré

(1.2.1) Le développement d'un programme passe par trois phases successives :

1) écriture et enregistrement des différents fichiers-source,
2) compilation séparée des fichiers `.cpp`, chacun d'eux donnant un *fichier-objet* portant le même nom, mais avec l'extension `.obj`,
3) lien des fichiers-objets (assurée par un programme appelé *linker*) pour produire un unique *fichier-exécutable*, portant l'extension `.exe` ; ce dernier pourra être lancé et exécuté directement depuis le système d'exploitation.

(1.2.2) Ce développement est grandement facilité lorsqu'on travaille dans un *environnement de développement intégré* (en abrégé : *EDI*), qui gère l'ensemble des fichiers et tâches à effectuer sous la forme d'un *projet*. Ce qui suit est une brève description de l'EDI *Visual C++* (version 6.0) de Microsoft.

La première chose à faire est de créer un nouveau projet, qu'on nomme par exemple `monproj`. On précise ensuite le type de projet. Pour l'instant nous nous contenterons de "Win 32 Console Application" (à l'exécution, les entrées de données se feront depuis le clavier et les affichages dans une fenêtre-texte de type DOS). Il faut savoir que Visual C++ permet de créer d'autres types de projets et propose tous les outils nécessaires à la construction de véritables applications Windows, avec gestion des fenêtres, menus, dialogues etc. (c'est un outil de développement extrêmement efficace pour qui maîtrise bien la programmation-objet).

L'EDI crée automatiquement sur le disque un répertoire appelé `monproj` destiné à regrouper tous les fichiers relatifs au projet.

L'espace de travail (*Workspace*) de Visual C++ est divisé en trois zones : à droite, la fenêtre d'édition permettant de taper les fichiers-source ; à gauche, la liste des fichiers-sources inclus dans le projet ; en bas, la fenêtre où s'affichent les messages du compilateur et du *linker*.

Après avoir créé les fichiers-source, on construit le projet (menu `build`, commande `build monproj.exe`), ce qui génère un programme exécutable appelé `monproj.exe`.

On peut alors demander l'exécution du programme (menu `build`, commande `execute monproj.exe`).

Remarque : l'espace de travail peut être sauvegardé à tout instant (dans un fichier dont l'extension est `.dsw`), ce qui permet de le réouvrir plus tard, lors d'une autre séance de travail.

Chapitre 2

ELEMENTS DU LANGAGE (1)

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.

On trouve parfois le mot `unsigned` précédant le nom d'un type entier (on parle alors d'entier *non signé* ; un tel entier est toujours positif).

(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 ↔ 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* (cf § 3.3).

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
```

(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.

2.2 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)
```

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

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`.

(2.2.4) **Conversion de type**

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).

On peut également provoquer une conversion de type grâce à l'opérateur `()` (*type casting*). Avec `x` comme ci-dessus, l'expression `(int)x` est de type `int` et sa valeur est la partie entière de `x`.

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

Ce sont les opérateurs `<<` et `>>`, utilisés en conjonction avec des 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*>

(2.2.6) **Formatage des sorties numériques**

On peut modifier l'apparence des sorties grâce aux expressions suivantes :

`endl`	provoque un passage à la ligne
`setfill(c)`	fixe le caractère de remplissage (si ce n'est pas un blanc)
`setprecision(p)`	fixe le nombre de chiffres affichés
`setw(n)`	fixe la largeur de l'affichage
`setbase(b)`	fixe la base de numération

Les quatre *manipulateurs* ci-dessus, sauf `setprecision()`, n'agissent que sur la prochaine sortie. Pour les utiliser, inclure la librairie `<iomanip.h>`.

Exemple :

```
cout << setbase(16) << 256 << endl;                                  // affiche 100 et passe à la ligne
cout << setprecision(5) << setfill('*') << setw(10) << 123.45678;    // affiche ****123.46
```

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 le programme : 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 définie avant d'être utilisée.

Dans un fichier-source `untel.cpp`, il est possible d'utiliser une fonction qui est définie dans un autre fichier. Mais, pour que la compilation s'effectue sans encombre, il faut en principe que cette fonction soit déclarée dans `untel.cpp` (en usant au besoin de la directive `#include`) : voir à cet égard (2.5.2).

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**

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**

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**

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);
```

2.5 Exemple de programme

(2.5.1) Voici un petit programme complet écrit en C++ :

```
// ------------------- fichier gazole.cpp ---------------------

#include <iostream.h>                 // pour les entrées-sorties

const double prix_du_litre = 0.89;    // déclaration du prix du litre de gazole (en euros) comme constante (hum!)
int reserve = 10000;                  // déclaration de la réserve de la pompe, en litres

double prix(int nb)                   // définition d'une fonction appelée "prix" :
                                      // cette fonction renvoie le prix de nb litres de gazole
{
        return nb * prix_du_litre;
}

int delivre(int nb)                   // définition d'une fonction appelée "delivre":  cette fonction renvoie 0
                                      // si la réserve est insuffisante, 1 sinon et délivre alors nb litres
{
        if (nb > reserve)             // instruction if :  voir paragraphe 2.6.1
                return 0;
        reserve -= nb;
        return 1;
}

int main()                            // définition de la fonction principale
{
        int possible;
        do                            // instruction do :  voir paragraphe 2.7.2
        {
                int quantite;
                cout << "Bonjour.  Combien voulez-vous de litres de gazole ?  ";
                cin >> quantite;
                possible = delivre(quantite);
                if (possible)
                        cout << "Cela fait " << prix(quantite) << " euros.\n";
        }
```

```
        while (possible);
        cout << "Plus assez de carburant.\n";
        return 0;                    // la fonction main renvoie traditionnellement un entier
}
```

(2.5.2) Comme le montre le programme précédent, il est tout-à-fait possible de n'écrire qu'un seul fichier-source `.cpp` contenant toutes les déclarations et instructions. Cependant, pour un gros programme, il est recommandé d'écrire plusieurs fichiers-source, chacun étant spécialisé dans une catégorie d'actions précise. Cette technique sera d'ailleurs de rigueur quand nous ferons de la programmation-objet.

A titre d'exemple, voici le même programme, mais composé de trois fichiers-sources distincts. On remarquera que le fichier d'en-tête est inclus dans les deux autres fichiers (voir (1.1.3)) :

```
// ------------------- fichier pompe.h ----------------------

const double prix_du_litre = 0.89;     // déclaration du prix du litre de gazole (en euros)

double prix(int nb);                   // déclaration de la fonction prix
int delivre(int nb);                   // déclaration de la fonction delivre


// ------------------- fichier pompe.cpp --------------------

#include "pompe.h"                     // pour inclure les déclarations précédentes

int reserve = 10000;                   // déclaration de la réserve de la pompe, en litres

double prix(int nb)                    // définition de la fonction prix
{
        .......  // comme en (2.5.1)
}

int delivre(int nb)                    // définition de la fonction delivre
{
        .......  // comme en (2.5.1)
}


// ------------------- fichier clients.cpp ------------------

#include <iostream.h>                  // pour les entrées-sorties
#include "pompe.h"                     // pour les déclarations communes

int main()                             // définition de la fonction principale
{
        .......  // comme en (2.5.1)
}
```

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";
}
```

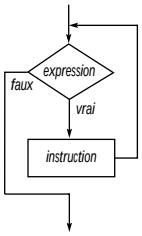
2.7 Boucles

(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 :



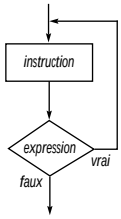
On remarque que le test est effectué avant la boucle.

(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) **Fonctions utiles**

Nous terminons en donnant quelques fonctions de la bibliothèque standard.

Fonctions mathématiques : elles sont déclarées dans `math.h`. Il y a notamment les fonctions suivantes, de paramètre `double` et de résultat `double` :
- `floor` (resp. `ceil`) : partie entière par défaut (resp. par excès),
- `fabs` : valeur absolue,
- `sqrt` : racine carrée,
- `pow` : puissance (`pow(x,y)` renvoie x^y),

- `exp`, `log`, `log10`,
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh` : fonctions trigonométriques.

Nombres aléatoires : il faut inclure `stdlib.h` et `time.h`. Alors :
- on initialise le générateur de nombres aléatoires (de préférence une seule fois) grâce à l'instruction `srand((unsigned) time(NULL));`
- ensuite, chaque appel de la fonction `rand()` donne un entier aléatoire compris entre 0 et `RAND_MAX` (constante définie dans `stdlib.h`).

Effacement de l'écran : instruction `system("cls");` (la fonction `system()` est déclarée dans `stdlib.h`).

Arrêt du programme : l'instruction `exit(1);` provoque l'arrêt immédiat du programme (la fonction `exit()` est déclarée dans `stdlib.h`).

Chapitre 3

ELEMENTS DU LANGAGE (2)

3.1 Enumérations

(3.1.1) *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`.

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

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

3.2 Tableaux

(3.2.1) *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
double tab[3];                // tableau de trois nombres réels
Vache troupeau[1000];         // tableau de mille vaches
```

(3.2.2) *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.

(3.2.3) 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]`.

(3.2.4) 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};
```

(3.2.5) **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 (3.2.3)
Phrase s;
```

(3.2.6) **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 !

3.3 Chaînes de caractères

(3.3.1) *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.*

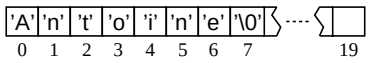
Exemples de déclarations :

```
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";
```

(3.3.2) 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 (3.2.6)
strcpy(prenom, "Antoine");     // correct :  voir ci-dessous (3.3.3)
```

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`.

(3.3.3) Dans la bibliothèque 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'`.

(3.3.4) **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';
```

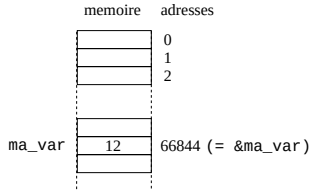
Remarque.— Il y a dans `ctype.h` quelques fonctions utiles concernant les caractères, entre autres :

`tolower()`	convertit une lettre majuscule en minuscule
`toupper()`	convertit une lettre minuscule en majuscule

3.4 Pointeurs, références

(3.4.1) **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` :



(3.4.2) **Pointeurs**

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

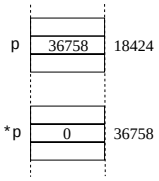
La déclaration d'un pointeur est de la forme suivante :

<*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
```

(3.4.3) 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
```

(3.4.4) **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
```

(3.4.5) **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*).

(3.4.6) **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]
```

A noter :

```
int U[5];
U = T;         // illégal :  U ne peut être modifié (pointeur constant) :  voir (3.2.6)
```

(3.4.7) Comme conséquence de (3.4.6), 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)`.

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

Une chaîne de caractères est habituellement déclarée :
- soit comme tableau de caractères, comme en (3.3) ; 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 (3.3.3). 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, 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, char *source)               // copie source dans dest
{
       char *temp = dest;
       while (*source)
              *dest++ = *source++;
       *dest = *source;              // copie le terminateur '\0'
       return temp;
}
```

(3.4.9) **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 int;`

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 (3.2.1)). 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`.

3.5 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.

Certains de ces opérateurs concerneront la programmation-objet.

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