


République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université des sciences et de la technologie d'Oran Mohammed Boudiaf
Faculté de Génie Électrique
Département d'Automatique

Polycopié de cours

Programmation en C++

DELLA KRACHAI Mohamed

Octobre 2019

Avant-propos

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.

Ces notes de cours sont destiné aux étudiants d'automatique, parcours licence L3. C'est un support confectionné en fonction du programme officiel de la formation.

Cette formation est une initiation à la prise en main de la programmation en langage C++. L'objectif est d'accompagner l'étudiant à travers des concepts simples et des exemples de programmes afin de maitriser les éléments du langage.

Della Krachai Mohamed

Table des matières

CHAPITRE 1

Présentation du langage C++

Plan de ce chapitre

I. Historique

C++ est un langage de programmation développé par Bjarne Stroustrup à partir de 1979 chez Bell Labs. C ++ s'exécute sur une variété de plates-formes, telles que Windows, Mac OS et les différentes versions d'UNIX.

C++ est un langage de programmation statique, compilé, à usage général, sensible à la casse, langage libre qui prend en charge la programmation procédurale, orientée objet et générique.

C++ est considéré comme un langage de niveau intermédiaire, car il combine les fonctionnalités du langage haut niveau et bas niveau.

C++ est très utilisé pour écrire des pilotes de périphériques et d'autres logiciels qui reposent sur la manipulation directe du matériel avec des contraintes en temps réel.

C++ est largement utilisé pour l'enseignement et la recherche.

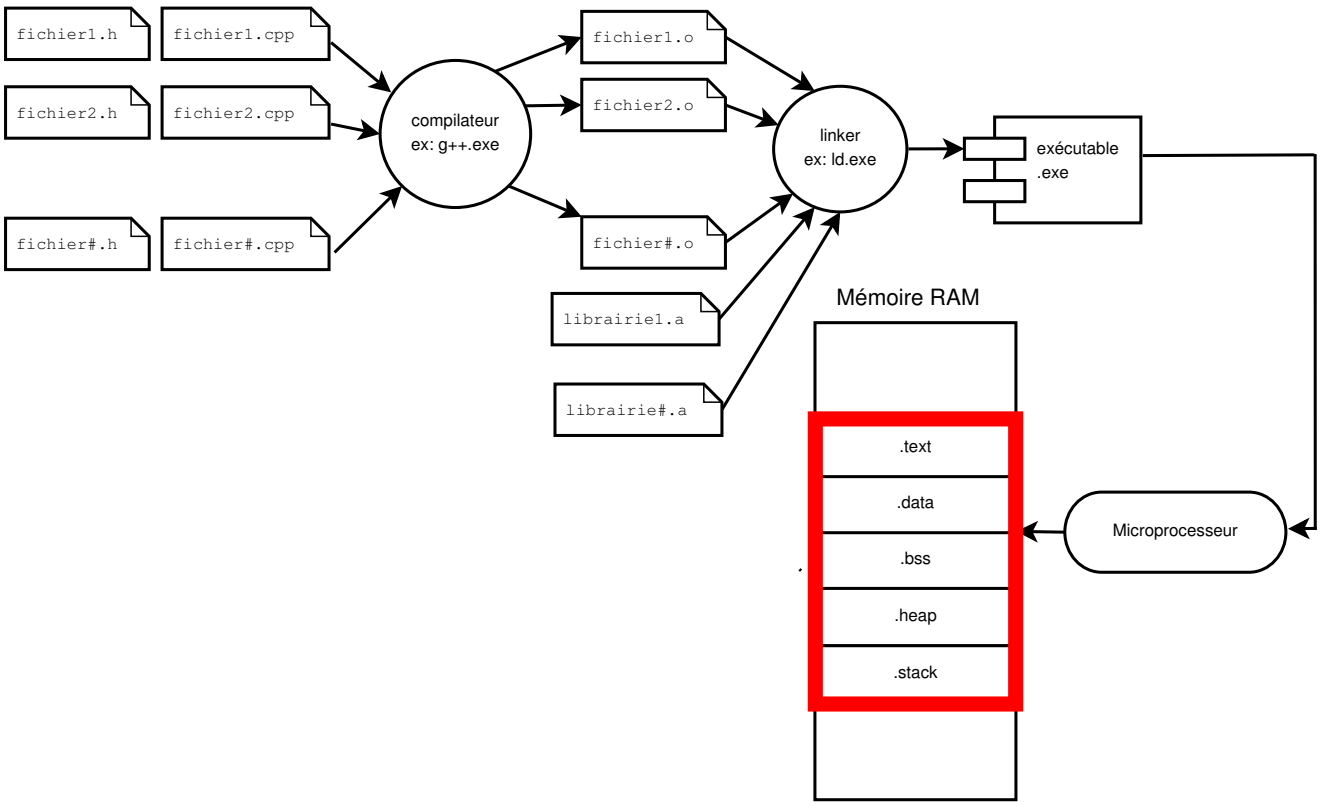


FIGURE 1.1 – Étapes de production d'un fichier exécutable à partir de sources

II. Phases de génération d'un exécutable

Généralement, un projet est décomposé en plusieurs unités de traitement (fichiers source). Chaque unité est écrite dans un fichier (.cpp) à part, dont le contenu représente une partie spécifique du traitement.

En premier lieu, le **compilateur** vérifie les erreurs syntaxiques dans le programme. S'il y a des erreurs, chacune est reportée en indiquant la ligne de l'instruction qui l'a provoquée et la nature de l'erreur. Après correction, le compilateur produit un code binaire (.o) correspondant au fichier source (.cpp).

D'autre part, le programmeur peut faire appel à un code binaire (.a ou .lib) réalisé par une tierce personne ou organisme.

L'éditeur de liens (**linker**), permet de réunir tout ces fichiers binaires et de le organiser en mémoire pour en produire un exécutable (.exe, .out, .bin, ...). Cet exécutable est dépendant de la machine (microprocesseur) et du système d'exploitation (windows, linux, Mac OS, ...) sur lesquels il a été généré.

Dans un ordinateur personnel (PC), l'architecture Von-Neumann est la plus utilisée. Dans cette architecture, le programme exécutable et les données sont rangées dans la même mémoire.

Lors de l'exécution d'un programme, le système d'exploitation réserve de la mémoire de la

manière suivante :

- Les instructions dans le segment mémoire **.text** appelé également code segment,
- Les variables non initialisées dans le segment mémoire **.bss**,
- Les variables initialisées dans le segment mémoire **.data**,
- Les variables locales et temporaires dans le segment mémoire pile **.stack**,
- Les variables dont l'allocation est dynamique dans le segment mémoire tas **.heap**.

L'utilitaire "**size.exe**" permet d'afficher l'occupation mémoire d'un exécutable.
Exemple :

```
C:\>size a.exe
   text    data     bss     dec     hex filename
 835212   17776    5344  858332   d18dc a.exe
```

III. Environnement de développement en C++

Plusieurs approches peuvent-être utilisées afin de développer des applications en C++. Cela dépend essentiellement des outils utilisés, qui peuvent être sous la forme d'environnement de développement integré **IDE** ou à la base de ligne de commande **CLI**.

III. 1. Éditeur de texte

Cela sera utilisé pour saisir le(s) programme(s) source. Les exemples d'éditeurs : NotePad++, Sublime Text, ...etc

Les fichiers source en C ++ ont l'extension .cpp (hpp), .cxx ou .cc.

Avant de commencer la programmation, il faut s'assurer d'avoir un éditeur approprié installé sur la machine. Les IDE intègrent directement un utilitaire de saisie de programme source.

III. 2. Compilateur C ++

Le compilateur le plus souvent utilisé est le compilateur GNU C / C ++. C'est un compilateur open-source, très utilisé dans le milieu universitaire et par les développeurs. Il est délivré sous licence **GPL**. Il est portable sur toutes les plateformes OS existantes.

III. 3. Installation du compilateur GNU C/C++

Pour installer GCC sous Windows, il faut d'abord installer MinGW depuis le site : `www.mingw.org`.

Proceder par la suite à l'installation dans un dossier ne contenant pas d'espace dans le chemin comme le montre la figure1.2 .
Ajoutez le sous-répertoire **bin** de votre installation MinGW à votre variable d'environnement **PATH** figure 1.3()afin que vous puissiez saisir ces outils sur la ligne de commande par leurs

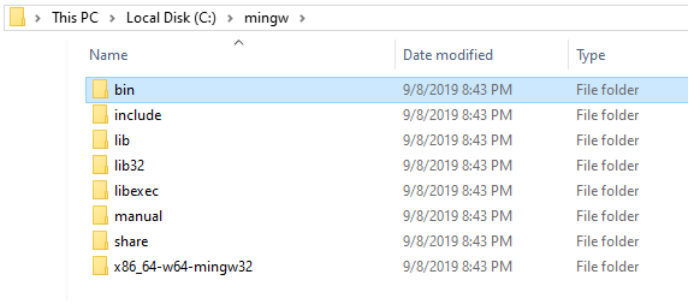


FIGURE 1.2 – Le dossier Mingw et son contenu

noms.

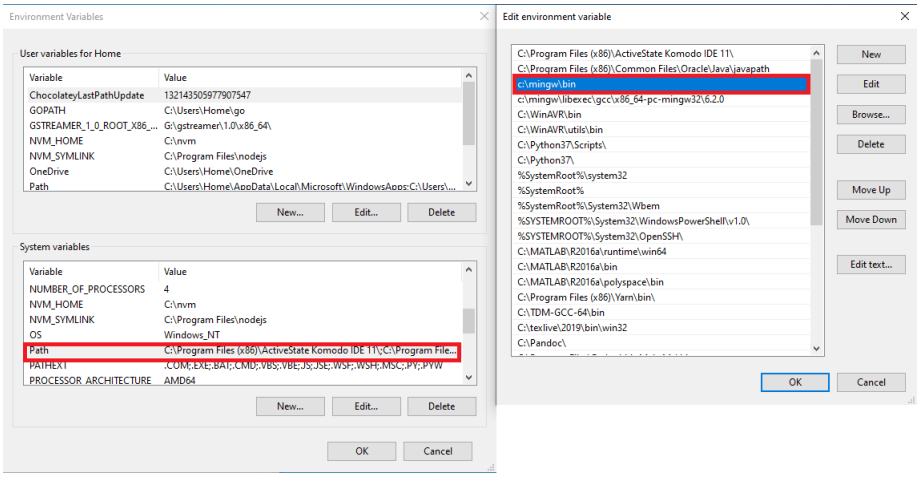


FIGURE 1.3 – Variable d'environnement ajoutée au systeme

Une fois l'installation terminée, vous pourrez exécuter gcc, g ++, ar, ranlib, dlltool et plusieurs autres outils GNU à partir de la ligne de commande de Windows. La figure 1.4 donne le résultat d'appel du compilateur avec l'option *version*. Cela confirme la phase d'installation.

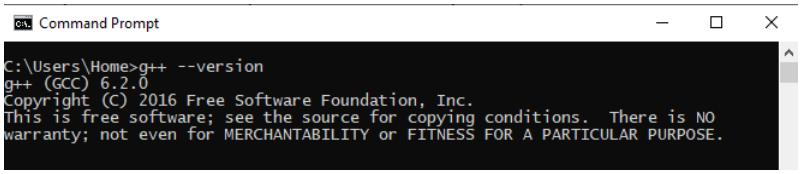


FIGURE 1.4 – Vérification du fonctionnement du compilateur

III. 4. Compiler et exécuter le programme C++

Nous allons maintenant voir comment enregistrer, compiler et exécuter le programme. Veuillez suivre les étapes ci-dessous :

1. Ouvrez un éditeur de texte et saisissez le code ci-dessus.

```cpp
#include <iostream>

int main() {

  //Imprimer la chaîne de caractères, "Bonjour!"
  std::cout << "Bonjour" << std::endl;

  return 0;
}
```

2. Enregistrez le fichier sous le nom : **prg1.cpp**
3. Ouvrez une invite de commandes (*touche windows* et tapez *cmd*) et allez dans le répertoire où vous avez enregistré le fichier (**C :\learn_cpp \chapitre1**).
4. Tapez **g++ prg1.cpp** et appuyez sur Entrée pour compiler votre code. S'il n'y a pas d'erreurs dans votre code, l'invite de commande vous amènera à la ligne suivante et générera un fichier exécutable a.exe.

   Tapez 'a.exe' pour exécuter votre programme.
   Vous pourrez voir (figure 1.5)'**Bonjour**' s'afficher sur la fenêtre.

FIGURE 1.5 – Etapes de compilation

À retenir
Assurez-vous que **g++** est dans votre path et que vous l'exécutez dans le répertoire contenant le fichier **prg1.cpp**.

IV. Débogage et recherche de problèmes

IV. 1. Déboguer un programme avec GDB

GDB fonctionne sur le principe d'une invite de commandes. Pour démarrer une session, il suffit de lancer GDB en précisant l'exécutable à débugger en paramètre :

```
gdb program
```

Une fois GDB lancé, des commandes peuvent-être saisies pour effectuer certaines opérations. Une ligne de commande GDB commence par le nom de la commande, qui peut être suivi par des paramètres.

```
$gdb programme -q
(gdb) commande params #comment
...
```

Pour déboguer un programme, il est nécessaire de pouvoir obtenir certaines informations le concernant, comme le nom et le type des variables, ainsi que les numéros de lignes correspondant aux instructions. Pour pouvoir les obtenir, on doit le préciser lors de la compilation. Il s'agit de l'option **"-g"**.

```
$ g++ -g programme.cpp
```

Pour lancer l'execution du programme (.exe), saisissez :

```
(gdb) run args
```

Pour stopper l'exécution, Il suffit d'utiliser la commande **kill** :

```
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb)
```

Pour contrôler l'exécution d'un programme, on utilise des points d'arrêt. On donne à GDB une liste d'endroits où il doit faire une pause dans l'exécution. Cela permet par exemple de vérifier l'état d'une variable ou le contenu d'un registre.

Il y a trois façons de préciser un point d'arrêt :

1. indiquer un endroit précis (fonction, ligne, fichier) ;
2. lors de la modification d'une variable (ou à sa lecture) ;
3. lorsqu'un certain évènement se produit (création d'un nouveau processus, chargement d'une librairie, throw ou catch d'une exception, etc.).

Respectivement, ces trois types de points d'arrêt se nomment breakpoints, watchpoints (points de surveillance), et catchpoints (points d'interception). Lorsque GDB interrompt le programme, il affiche la raison de l'arrêt, la ligne de l'instruction suivante (la prochaine qui sera exécutée) et l'invite de commande :

```
(gdb) r
Starting program: main
Breakpoint 3, main () at sample1.cpp:9
9           a = a / b;
(gdb)
```

IV. 2. Environnement de dévelopement integré

Un environnement de dévelopement integré ou IDE (Integrated Development Environment) est un environnement convivial disposant de toutes les fonctionnalités de développement et exécution de programmes et de projets. Il intègre des menus et des outils afin de réaliser :

— L'édition des programmes sources,

— La compilation,

— La production de l'exécutable, des librairies, ...
— Le débogage et la recherche des erreurs,
— L'organisation des fichiers sources (.cpp, .h) sous forme de projet.
— Des fonctionnalités avancées.

CodeBlocks [1], Eclipse [2], Microsoft visual studio [3], Visual Studio Code [4] représentent les IDE les plus célèbres. Tout IDE comporte un ensemble de volets :

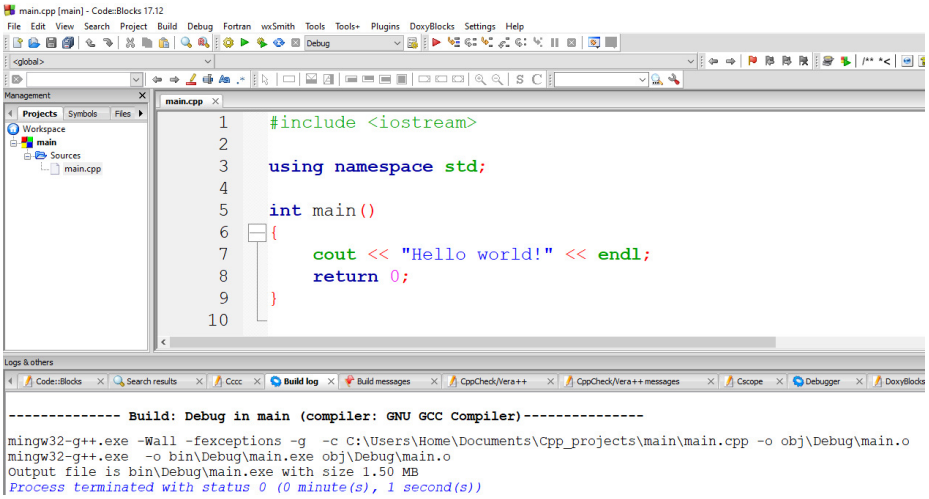


FIGURE 1.6 – IDE CodeBlocks

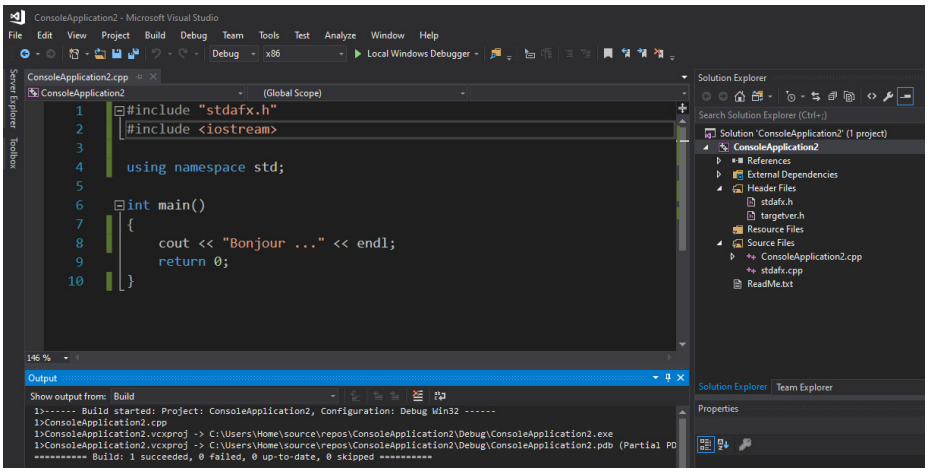


FIGURE 1.7 – IDE Visual Studio Solution

— volet explorateur de projet : cette partie affiche l'arborescence du projet (fichiers source, en-tête, libraries, binaires, ...),
— volet editeur : l'utilsateur saisi son code source dans cet espace. La saisie du programme est conviviale et propose une *autocompletion*, la coloration des mots cléfs, la suggestion en fonction de la première lettre saisie, et d'autres fonctionnalités.

1. `www.codeblocks.org`
2. `www.eclipse.org`
3. `www.microsoft.com`
4. `www.microsoft.com`

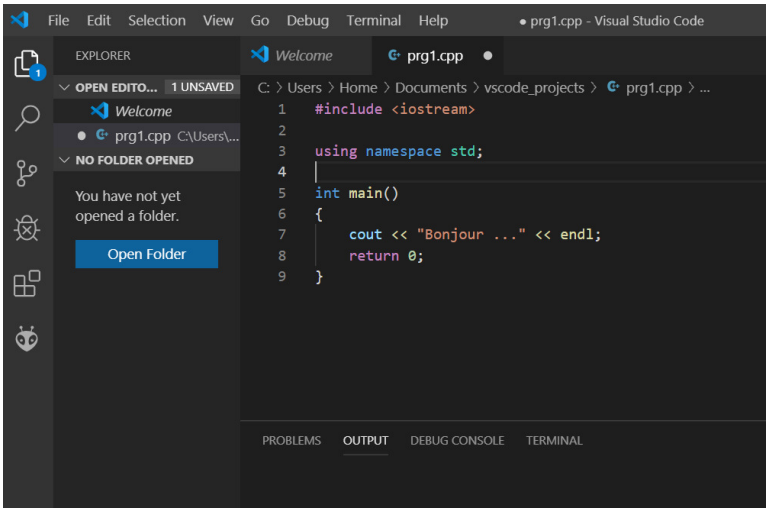


FIGURE 1.8 – IDE Visual Studio Code

— volet résultat de compilation : dans lequel les erreurs (s'il y en a) sont affichées, la commande de compilation et ses resultats.

— Des barres d'outils incluants les boutons de compilation, debuggage, ...

CHAPITRE 2

Syntaxe élémentaire en langage C++

Plan de ce chapitre

I. Introduction

Dans ce cours les points suivants sont abordés :

— La manipulation des données (variables et constantes),
— Les règles de conversion entre données,
— Les opérateurs et les opérations sur les données.

I. 1. Premier programme en C++

Un programme est une suite d'instructions réalisant une tâche bien définie.

Une instruction se termine par un point-virgule **";"**. Elle peut servir à déclarer un objet, lui donner une valeur, appeler une fonction, appeler plusieurs fonctions de façon imbriquée, etc.

Un bloc d'instructions est délimité par des accolades **...**. Toute instruction fait naturellement partie d'un bloc à un certain niveau (par exemple elle fait partie d'une boucle ou d'une fonction).

Une variable déclarée à l'intérieur d'un bloc est automatiquement détruite à la sortie du bloc. On ne peut alors plus l'utiliser. Si l'on veut que la variable continue d'exister après la fin de ce bloc, il suffit de la déclarer avant ce bloc.

Le listing suivant présente un programme assez simple :

```
1  #include <iostream.h>
2
3  int main() {
4
5    //Imprimer la chaîne de caractères, "Bonjour!"
6    std::cout << "Mon premier programme en C++" << std::endl;
7
8    return 0;
9
10 }
```

II. Variables et Mémoire

On distingue les mémoires de stockage durables (comme les disques durs) et les mémoires de travail temporaires (mémoire **vive** ou RAM -Random Access Memory). Cette dernière est utilisée pour conserver les données et les résultats des calculs des programmes.

Dans cette section nous allons découvrir comment un ordinateur stocke et gère les variables à l'intérieur de la mémoire vive.

La RAM est scindée en deux parties comme le montre la figure 2.1 :

1. Les **adresses** : une adresse est un nombre qui permet à l'ordinateur de se repérer dans la mémoire. On commence à l'adresse 0 (au tout début de la mémoire) et on finit à une adresse de terminaison. Le nombre d'adresses dépend de la taille de la RAM ;
2. Les **données** : à chaque adresse, on peut stocker une valeur (un nombre). On ne peut stocker qu'un nombre par adresse.

III. Déclarer une variable

La déclaration informe le compilateur sur les types de variables qui sont utilisés dans le programme.
Pour déclarer une variable, on précise son **type** suivi de son **identificateur** (son nom).

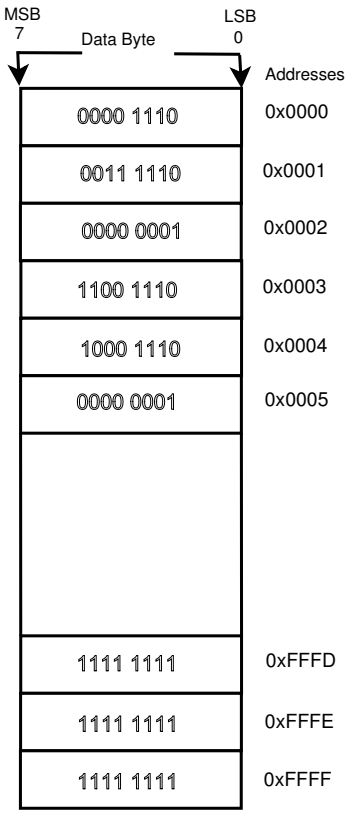


FIGURE 2.1 – L'organisation de la mémoire

Pour créer une variable,la syntaxe est la suivante :

```
TYPE IDENTIFICATUR {VALEUR};
```

III. 1. Les identificateurs

L'identificateur d'une variable est composé d'un ensemble de chiffres ou de lettres en respectant les règles suivantes :

- Le premier caractère doit être **obligatoirement** une **lettre**,
- Les minuscules ou les majuscules (la **casse**) sont **autorisées** et considérées comme **différentes**,
- Le caractère "_" (underscore) est autorisé,
- Il faut veuiller à ne pas utiliser des **mots-clés** ou **mots réservés** du langage (tableau 2.1).

```
1    int x;           // créer une variable sans l'initialiser
2    int x = 123;     // créer une variable et l'initialiser avec la
        valeur 123
3    int x(123);      // créer une variable et l'initialiser avec la
        valeur 123
```

Attention à :

```
1    int x();           // déclarer une fonction
```

Le tableau suivant resume les mots cles :

asm	auto	bool	break	case	catch	char
class	const	const_ cast	continue	default	delete	do
double	dynamic_ cast	else	enum	explicit	extern	false
float	for	friend	goto	if	inline	int
long	mutable	namespace	new	operator	private	protected
public	register	reinterpret_ cast	return	short	signed	sizeof
static	static_ cast	struct	switch	template	this	throw
true	try	typedef	typeid	typename	union	unsigned
using	virtual	void	volatile	while		

TABLE 2.1 – Les mots clés utilisés en C++

III. 2. Les constantes

On parle de constante dans le cas ou on n'a pas besoin de modifier la valeur d'une variable.

```
1  #include <iostream>
2
3  int main() {
4      const int x { 123 };
5      x = 456; // erreur
6  }
```

Ce programme produit l'erreur suivante :

```
error: assignment of read-only variable 'x'
```

Le type définit le nombre d'octets mémoire que prendra la variable.

Les types de données de base sont :

- int : valeur entière
- char : caractère simple
- float : nombre réel en virgule flottante
- double : nombre réel en virgule flottante double précision.

Il est possible de connaître le nombre d'octets utilisé pour représenter un type ou une variable, en utilisant l'opérateur **sizeof**, qui prend en paramètre la variable ou le type. Par exemple, pour utiliser sizeof avec des types :

```
1  #include <iostream>
2  using namespace std;
3  int main() {
4      cout << "sizeof(int) = " << sizeof(int) << endl;
5      cout << "sizeof(double) = " << sizeof(double) << endl;
6      cout << "sizeof(bool) = " << sizeof(bool) << endl;
7      cout << "sizeof(char) = " << sizeof(char) << endl;
8  }
```

Ce programme donne en sortie :

```
sizeof(int) = 4
sizeof(double) = 8
sizeof(bool) = 1
sizeof(char) = 1
```

Des qualificateurs comme **short**, **signed**, **unsigned** peuvent enrichir les types de données.

IV. Portée d'une variable (Scope)

La portée d'une variable est définit comme étant son existance à partir du moment de sa déclaration dans un bloc jusqu'à la sortie de ce bloc.

```
1  #include <iostream>
2
3  int main() {
4      std::cout << x << std::endl;
5      const int x { 123 };
6  }
```

produit le résultat suivant :

```
main.cpp: In function 'int main()':
main.cpp:4:18: error: 'x' was not declared in this scope
     std::cout << x << std::endl;
                  ^
```

Ce qui signifie "x n'est pas déclarée dans cette portée".

V. Les entiers

Les entiers existent en trois types :

1. **short int**,
2. **int**,
3. **long int**.

Le tableau 2.2 résume l'occupation mémoire pour chaque type d'entiers.
Chaque type peut-être signé(**signed**) ou non signé(**unsigned**).

Le type char est aussi un type entier, il manipule les caractères en code **ASCII**.

VI. Les réels

Ils s'expriment sous trois types différents :

1. **float** (flottant),
2. **double**,
3. **long double**.

Type	Limite inférieure	Limite supérieure	Taille en octets
char	-128	127	1
unsigned char	0	255	1
short int	-32768	32767	2
unsigned short int	0	65535	2
int	-2 147 483 648	2 147 483 647	4
unsigned int	0	4 294 967 295	4
long int	-2 147 483 648	2 147 483 647	4
unsigned long int	0	4 294 967 295	4

TABLE 2.2 – Plage couverte par les entiers

Type	Limite inférieure	Limite supérieure	Taille en octets
float	$-3.4 \cdot 10^{-38}$	$3.4 \cdot 10^{38}$	4
double	$-1.7 \cdot 10^{-308}$	$1.7 \cdot 10^{308}$	8
long double	$-3.4 \cdot 10^{-4932}$	$3.4 \cdot 10^{4932}$	10

TABLE 2.3 – Plage couverte par les réels

Le C++ propose un ensemble de fonctionnalités dans le fichier d'en-tête **limits**.
Ces fonctionnalités sont des classes génériques, qui utilisent une syntaxe avec des chevrons <>. La syntaxe générale est la suivante :

```
std::numeric_limits<TYPE>::FONCTION()
```

Avec **TYPE**, qui correspond au type pour lequel on veut obtenir des informations(par exemple **int**, **double**, etc), et **FONCTION**, qui correspond à l'information souhaitée (par exemple **max** pour la valeur maximale et **min** pour la valeur minimale).

```
1  #include <iostream>
2  #include <limits>
3  using namespace std;
4  int main()
5  {
6   cout << "Types entiers :" << endl;
7   cout << "Max(short int) = " << numeric_limits<short int>::max() <<
       endl;
8   cout << "Max(unsigned short int) = " << numeric_limits<unsigned
       short int>::max()<<endl;
9   cout << "Max(int) = " << numeric_limits<int>::max() << endl;
10  cout << "Max(unsigned int) = " << std::numeric_limits<unsigned int
       >::max() << endl;
11  cout << "Max(long int) = " << std::numeric_limits<long int>::max()
        << endl;
12  cout << "Max(unsigned long int) = " << std::numeric_limits<
       unsigned long int>::max() << endl;
13  cout << "Max(long long int) = " << std::numeric_limits<long long
       int>::max() << endl;
14  cout << "Max(unsigned long long int) = " << std::numeric_limits<
       unsigned long long int>::max() << endl;
15  cout << std::endl << "Types réels:" << endl
```

```
16  cout << "Max(float) = " << std::numeric_limits<float>::max() <<
        endl;
17  cout << "Max(double) = " << std::numeric_limits<double>::max() <<
        endl;
18  cout << "Max(long double) = " << std::numeric_limits<long double
        >::max() << endl;
19  }
```

Ce programme produit le resultat suivant :

```
Types entiers :
Max(short int) = 32767
Max(unsigned short int) = 65535
Max(int) = 2147483647
Max(unsigned int) = 4294967295
Max(long int) = 2147483647
Max(unsigned long int) = 4294967295
Max(long long int) = 9223372036854775807
Max(unsigned long long int) = 18446744073709551615

Types reels:
Max(float) = 3.40282e+038
Max(double) = 1.79769e+308
Max(long double) = 1.18973e+4932
```

VII. Règles de conversion implicites

Dans une expression mathématiques, où les *opérandes* sont de types différents, C++ applique quelques règles de conversion dites **implicites** car elles sont déterminées par le compilateur :

- Si un des opérandes est de type **long double**, les autres seront convertis en **long double**,
- Si un des opérandes est de type **double**, les autres seront convertis en **double**,
- Si un des opérandes est de type **float**, les autres seront convertis en **float**,
- Si un des opérandes est de type **unsigned long**, les autres seront convertis en **unsigned long**,
- Si un des opérandes est de type **long**, les autres seront convertis en **long**,
- Si un des opérandes est **unsigned**, les autres seront convertis en **unsigned**.

Exemple :

```
1  #include <iostream>
2  using namespace std;
3    int main()
4  {
5    int   x = 10.125;
6    float y = 10.125;
7    int   z = 2;
8    cout << "x = " << x << endl;
```

```
9       x = x + 0.5;
10      cout << "x + 0.5 = " << x << endl;
11      cout << "y = " << y << endl;
12      cout << "z = " << z << endl;
13      cout << "y + z = " << y+z << endl;
14    return 0;
15    }
```

```
Ce qui donne:
x = 10
x + 0.5 = 10
y = 10.125
z = 2
y + z = 12.125
```

VIII. Règles de conversion explicites

Ces règles sont appelées opérations de **cast**. Elles permettent une modification forcée du type de donnée.

```
x = (int)2.178; ou,
x = int(2.178);

Le résultat obtenu est:
            x = 2
```

IX. Les opérateurs et la priorité des opérations

Un opérateur est un symbole qui indique au compilateur d'effectuer des fonctions mathématiques ou logiques spécifiques. Le langage C++ intègre plusieurs opérateurs et fournit les types d'opérateurs suivants :

— Opérateurs arithmétiques

— Opérateurs d'assignation

— Opérateurs de comparaison

— Opérateurs bit à bit

— Opérateurs d'affectation

Les opérateurs sont soit **unaires**, soit **binaires**. Les opérateurs unaires ne prennent qu'une seule opérande. Les opérateurs binaires prennent toujours deux opérandes, une avant et une après (exemple : 2+5) l'opération.

IX. 1. Opérateurs arithmétiques

Le tableau suivant présente tous les opérateurs arithmétiques supportés par le langage C++. Supposons que la variable A contenant la valeur 10 et que la variable B contenant la valeur 20.

Opérateur	**Description**	**Exemple**
+	La somme	$A+B=30$
-	La soustraction	$A-B=10$
$\star$	La multiplication	$A\star B=200$
/	La division	$B/A=2$
%	Modulo c'est le reste d'une division entier	$B\%A=0$
++	Opérateur d'incrémentation qui consiste à ajouter 1 à une variable	$A++$
−−	Opérateur de décrémentation qui consiste à retirer 1 à une variable.	$A--$

TABLE 2.4 – Les opérateurs arithmétiques

IX. 2. Les opérateurs d'assignation

Ces opérateurs permettent de simplifier des opérations telles que ajouter une valeur a une variable et stocker le résultat dans la meme variable. Une telle opération s'écrirait habituellement de la façon suivante par exemple : $x = x+2$

Avec les opérateurs d'assignation il est possible d'écrire cette opération sous la forme suivante : $x+=2$
Ainsi, si la valeur de x était 7 avant opération, elle sera de 9 après.

Les autres opérateurs du même type sont les suivants :

+=	additionne deux valeurs et stocke le résultat dans la variable (à gauche)
-=	soustrait deux valeurs et stocke le résultat dans la variable
***=**	multiplie deux valeurs et stocke le résultat dans la variable
/=	divise deux valeurs et stocke le résultat dans la variable

TABLE 2.5 – Les opérateurs d'assignation

IX. 3. Les opérateurs de comparaison

Les opérateurs de comparaison renvoient soit 1 ou 0, si le résultat de la comparaison est vrai ou faux.

==	Compare deux valeurs et vérifie leur égalité	x==3 Retourne 1 si x est égal à 3, sinon 0
<	opérateur d'infériorité stricte	x<3 Retourne 1 si x est inférieur à 3, sinon 0
<=	opérateur d'infériorité	x<=3 Retourne 1 si x est inférieur ou égal à 3, sinon 0
>	opérateur de supériorité stricte	x>3 Retourne 1 si x est supérieur à 3, sinon 0
>=	opérateur de supériorité	x>=3 Retourne 1 si x est supérieur ou égal à 3, sinon 0
!=	opérateur de différence	x!=3 Retourne 1 si x est différent de 3, sinon 0

TABLE 2.6 – Les opérateurs de comparaison

IX. 4. Les opérateurs logiques

Ce type d'opérateur permet de vérifier si plusieurs conditions sont vraies :

||	OU logique	((condition1)||(condition2))
&&	ET logique	((condition1)&&(condition2))
!	NON logique	(!condition)

TABLE 2.7 – Les opérateurs logiques

IX. 5. Les opérateurs bit-à-bit

Ces opérateurs traitent ces données selon leur représentation binaire mais retournent des valeurs numériques standard dans leur format d'origine.

Les opérateurs suivants effectuent des opérations bit-à-bit, c'est-à-dire avec des bits de même poids.

&	ET bit-à-bit	9 & 12 (1001 & 1100)	8 (1000)
|	OU bit-à-bit	9 | 12 (1001 | 1100)	13 (1101)
∧	OU exclusif bit-à-bit	9 ∧ 12 (1001 ∧ 1100)	5 (0101)

TABLE 2.8 – Les opérateurs bit-à-bit

IX. 6. Les opérateurs de décalage de bit

Ce type d'opérateur traite ses opérandes comme des données binaires d'une longueur de 32 bits.

Les opérateurs suivants effectuent des decalages sur les bits, c'est-à-dire qu'ils décalent chacun des bits d'un nombre de bits vers la gauche ou vers la droite. La première opérande désigne la donnée sur laquelle on va faire le décalage, la seconde désigne le nombre de bits duquel elle va être décalée.

La priorité des opérateurs spécifie l'ordre des opérations des expressions qui contiennent

≪	Décalage à gauche	6 ≪ 1 (110 ≪ 1)	12 (1100)
≫	Décalage à droite	6 ≫ 1 (0110 ≫ 1)	3 (0011)

TABLE 2.9 – Les opérateurs de décalage de bit

plusieurs opérateurs. L'associativité des opérateurs spécifie si, dans une expression contenant plusieurs opérateurs ayant la même priorité, un opérande est groupé avec celui situé à sa gauche ou celui situé à sa droite. Le tableau suivant indique la priorité des opérateurs C++ (de priorité décroissante).

()	[]										
– –	++		~								
*	/	%									
+	–										
«	»										
<	<=	>=	>								
==	!=										
ˆ	&	|									
&&	||										
? :											
=	+=	-=	*=	/=	%=	«=	>>=	>>>=	&=	≙	|=
,											

TABLE 2.10 – Priorité des opérateurs

CHAPITRE 3

Structures conditionnelles et boucles

Plan de ce chapitre

Les structures conditionnelles exigent que le programmeur spécifie une ou plusieurs conditions à évaluer ou à tester par le programme, et consiste à décider l'ordre d'exécution des déclarations en fonction de certaines conditions ou à répéter un groupe d'instruction jusqu'à ce que certaines conditions spécifiées soient remplies.

Voir ci-dessous la figure qui montre la forme générale d'une structures conditionnelles typique trouvée dans la plupart des langages de programmation.

— **Instruction if** : Consiste en une expression booléenne suivie d'une ou plusieurs instructions.

— **Instruction if…else** : L'instruction "if" peut être suivie d'une instruction optionnelle "else", qui s'exécute lorsque l'expression booléenne est **false**.

— **Instructions imbriqués if** : On peut utiliser une instruction "if" ou "else if" dans une autre instruction "if " ou "else if".

— **Instruction switch** : L'instruction switch permet de tester l'égalité d'une variable à une liste de valeurs.

— **Déclaration imbriqués switch** : Vous pouvez utiliser une instruction switch dans une autre instruction switch (s).

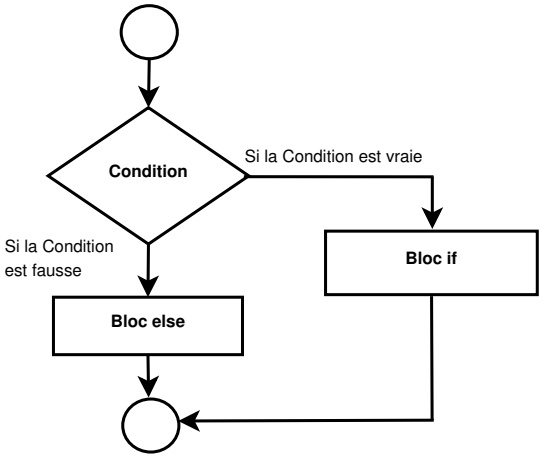


FIGURE 3.1 – Déclaration **if...else**

I. Differentes formes de l'instruction if

I. 1. Instruction if

La forme générale d'une déclaration **if** est :

```
if( expression )
{
  Declaration-interieur;
}

Declaration-exterieur;
```

Si l'expression est vraie, alors 'Declaration-interieur' elle sera exécutée, sinon 'Declaration-interieur' est ignoré et seulement 'Declaration-exterieur' est exécuté.
Exemple :

```
#include <iostream>
using namespace std;

void main()
{
      int x,y;
      x=15;
      y=13;
      if (x > y )
      {
        cout << "x est supérieur a y" <<endl;
      }
}
```

I. 2. Instruction if. . . else

La forme générale d'une instruction **if... else** est :

```
if( expression )
{
  Bloc-instructions-1;
}else
{
  Bloc-instructions-2;
}
```

Si l'expression est vraie, le '*Bloc-instructions-1*' est exécuté, sinon '*Bloc-instructions-1*' est ignoré et '*Bloc-instructions-2*' est exécuté.

Exemple :

```
#include <iostream>
using namespace std;

void main()
{
      int x,y;
      x=14;
      y=19;
      if (x > y )
      {
       cout << "x est supérieur a y" <<endl;
      }
      else
      {
       cout << "y est supérieur a x"<<endl;
      }
}
```

I. 3. Instructions imbriqués if

La forme générale d'une instruction imbriquée if ... else est,

```
if( expression )
{
  if( expression1 )
   {
      Bloc-instructions1;
   }
  else
   {
     Bloc-instructions2;
   }
}else
{
 Bloc-instructions3;
}
```

Si '*expression*' est fausse, 'Bloc-instructions3' sera exécuté, sinon il continue à effectuer le test pour 'expression 1'. Si l'expression 1 est vraie, 'Bloc-instructions1' est exécutée sinon

'Bloc-instructions2' est exécutée.

Exemple :

```
#include <conio.h>
#include <iostream>
using namespace std;

void main( )
{
      int a,b,c;
      clrscr();
      cout << "entrez 3 nombres"<<endl;
      cin >> a >> b >> c;
      if(a>b)
      {
       if( a > c)
       {
         cout << "a est le plus grand"<<endl;
       }
       else
       {
         cout << "c est le plus grand"<<endl;
       }
      }
      else
      {
       if( b> c)
        {
          cout << "b est le plus grand"<<endl;
        }
       else
        {
          cout << "c est le plus grand"<<endl;
        }
       }
     getch();
}
```

I. 4. Instruction else if

La forme générale de else if est,

```
if(expression1)
{
 Bloc-instructions1;
}else if(expression2)
{
 Bloc-instructions2;
}else if(expression3 )
{
 Bloc-instructions3;
```

```
}else
 Bloc-instructions4;
```

L'expression est testée à partir du haut vers le bas. Dès que l'état réel est trouvé, l'instruction qui lui est associée est exécutée.

Exemple :

```
#include <conio.h>
#include <iostream>
using namespace std;

void main()
{
      int a;
      cout << "entrez un nombre"<<endl;
      cin >> a;
      if( a%5==0 && a%8==0)
      {
        cout <<"divisible par les 5 et 8"<<endl;
      }
      else if( a%8==0 )
      {
        cout <<"divisible par 8"<<endl;
      }
      else if(a%5==0)
      {
        cout <<"divisible par 5"<<endl;
      }
      else
      {
        cout <<"non divisible"<<endl;
      }
     getch();
}
```

I. 5. L'opérateur ?:

Nous avons couvert l'opérateur conditionnel **?** : Dans le chapitre précédent, qui peut être utilisé pour remplacer les instructions if ... else. Il a la forme générale suivante :

```
Exp1 ? Exp2 : Exp3;
```

Exp1, Exp2 et Exp3 sont des expressions. Si Exp1 est vrai donc Exp2 sera exécutée sinon Exp3 qui sera exécutée.

À retenir

1. Dans une déclaration if, si on a une seule instruction à l'intérieur donc vous n'êtes pas obligé de fermer les accolades

```
int a = 5;
if(a > 4)
 cout<<"success";
```

2. == doit être utilisé pour la comparaison dans l'expression if , si vous utilisez l'expression = renverra toujours true, car elle effectue l'affectation et non la comparaison.
3. Toutes les valeurs sont considérées comme vraies sauf 0 (zéro).

```
if(45)
 cout<<"bonjour";
```

II. switch, case, default

L'instruction "switch" effectue un test logique multi-états : "Selon la variable, si elle est à telle valeur alors faire ceci, si elle est à telle autre valeur faire cela, si elle est à encore une autre valeur faire autre chose, ..., sinon : faire le traitement par défaut.".

Syntaxe :

```
switch(<Variable>)
{
    case <Valeur1>:
        <ActionsSiValeur1>;
        break;
    case <Valeur2>:
        {
            <ActionsSiValeur2>;
        }
        break;
    case <Valeur3>:
    case <Valeur4>:
        {
            <ActionsSiValeur3Ou4>;
        }
        break;
    case <Valeur5>:
        {
            <ActionsSiValeur5>;
        }
        break;
    ///...
    case <ValeurN>:
        {
            <ActionsSiValeurN>;
        }
```

```
        break;
    default:
        {
            <TraitementParDéfaut>
        }
        break;
}
```

Où <Variable> est la variable à tester <Valeur1> à <ValeurN> sont les différentes valeurs que <Variable> peut prendre et qui nécessitent un traitement particulier, <ActionsSiValeur1> à <ActionsSiValeurN> sont les différents traitements pour chaque valeur nécessitant leur propre traitement, <ActionsSiValeur3Ou4> est un cas où plusieurs valeurs (ici <Valeur3> et <Valeur4>) nécessitent le même traitement (remarquez l'absence de break; entre les deux cas, autrement obligatoire), <TraitementParDéfaut> signalé par la clause default est le traitement réservé à toute autre valeur que celles renseignées par les clauses case. Il est à noter que la clause default n'est pas obligatoire. Par contre il est impératif de mettre une clause break après chaque clause case suivie de code.

Dans un souci d'optimisation de code et de compréhension les valeurs renseignées dans les clauses case doivent décrire les cas uniques nécessitant des traitements personnalisés. La clause default traitant tous les autres cas.

Exemple :

```
#include <iostream>

using namespace std;

int main (int argc, char* argv[])
{
    int vInteger = argc;

    switch (vInteger)
    {
        case 0:
            {
                cout << "vInteger = 0" << endl;
            }
            break;
        case 1:
        case 2:
            {
                cout << "vInteger = 1 ou 2" << endl;
            }
            break;
        default:
            {
                cout << "vInteger != (0, 1 et 2)" << endl;
            }
            break;
    }
    return 0;
}
```

III. Les commandes de rupture de séquence

Il existe d'autres commandes de rupture de séquence (c'est-à-dire de changement de la suite des instructions à exécuter). Ces commandes sont les suivantes :

```
continue;
ou
break;
ou
return [valeur];
```

return permet de quitter immédiatement la fonction en cours. Comme on l'a déjà vu, la commande return peut prendre en paramètre la valeur de retour de la fonction.

break permet de passer à l'instruction suivant l'instruction while, do, for ou switch la plus imbriquée (c'est-à-dire celle dans laquelle on se trouve).

continue saute directement à la dernière ligne de l'instruction while, do ou for la plus imbriquée. Cette ligne est l'accolade fermante. C'est à ce niveau que les tests de continuation sont faits pour for et do, ou que le saut au début du while est effectué (suivi immédiatement du test). On reste donc dans la structure dans laquelle on se trouvait au moment de l'exécution de continue, contrairement à ce qui se passe avec le break.

```
1 /* Calcule la somme des 1000 premiers entiers pairs : */
2 somme_pairs=0;
3 for (i=0; i<1000; i=i+1)
4 {
5     if (i % 2 == 1) continue;
6     somme_pairs=somme_pairs + i;
7 }
```

IV. Les Boucles

Vous pouvez rencontrer des situations, quand un bloc de code doit être exécuté plusieurs fois. En général, les instructions sont exécutées séquentiellement.

Dans tous les langages de programmation, les boucles sont utilisées pour exécuter un ensemble d'instructions plusieurs fois jusqu'à ce qu'une condition particulière soit satisfaite. Une séquence d'instructions est exécutée jusqu'à ce qu'une condition soit vraie. Cette séquence d'instructions à exécuter est à l'intérieur des accolades appelées le corps de la boucle. Après chaque exécution du corps de la boucle, la condition est vérifiée, et si elle est trouvée"true", le corps de la boucle est exécuté à nouveau. Lorsque la vérification retourne "false", le corps de la boucle ne sera pas exécuté.

Le langage de programmation C++ fournit différents façon pour déclarer des boucles :

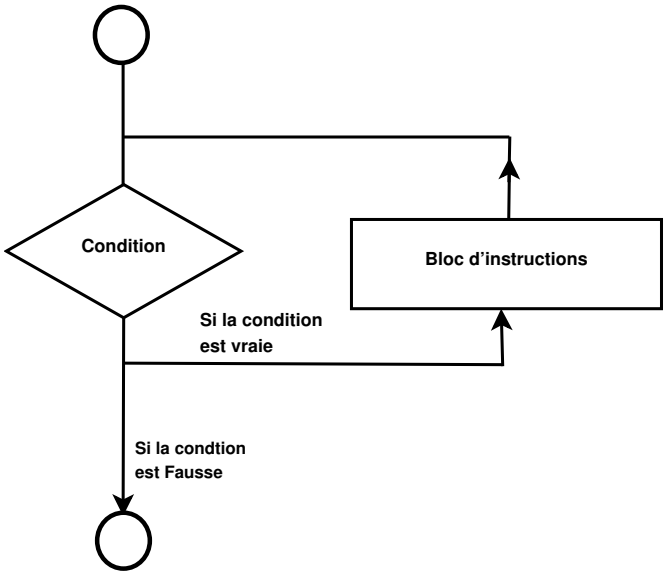


FIGURE 3.2 – Structure d'une boucle

— **La boucle while** : Répète une instruction ou un groupe d'instruction jusqu'à ce qu'une condition donnée est vraie. Il teste la condition avant d'exécuter le corps de la boucle.
— **La boucle for** : Exécute une séquence d'instructions plusieurs fois avec un code abrégé.
— **La boucle do...while** : Il ressemble à la déclaration while, sauf qu'elle teste la condition à la fin du corps de la boucle.
— **Les boucles imbriquées** : Vous pouvez utiliser une ou plusieurs boucles à l'intérieur d'une autre boucle while, for ou do..while

IV. 1. La boucle while

Tandis que la boucle while peut être traitée comme une boucle de contrôle. Il est complété en 3 étapes :

1. Initialisation de la variable (par exemple, $int x = 0;$)
2. Condition (par exemple $while(x <= 100)$)
3. Incrémentation ou décrémentation de la variable ($x++$ ou $x--$ ou $x = x+4$)

La syntaxe :

```
initialisation de la variable;
while (condition)
{
 bloc d'instructions;
 incrementation ou decrementation de la variable;
}
```

Exemple : Programme qui affiche les 10 premiers nombres naturels.

```
1 #include <iostream>
2 using namespace std;
3
4 void main( )
5 {
```

```
      int x;
      x=1;
      while(x<=10)
      {
        cout << x <<endl;
        x++;
      }
      getch();
}
```

IV. 2. La boucle for

La boucle for est utilisé pour exécuter un ensemble d'instructions de façon répétitive jusqu'à ce qu'une condition particulière soit satisfaite. Le format général est :

```
for(initialisation; condition ; incrementation/decrementation)
{
   Bloc d'instructions;
}
```

Dans la boucle for nous avons exactement deux points-virgules, un après l'initialisation et la deuxième après la condition. Dans cette boucle, nous pouvons avoir plus d'une initialisation ou une incrémentation ou décrémentation séparés par des virgules. Une boucle **for** ne peut avoir qu'une seule condition.

Exemple :

```
#include <iostream>
using namespace std;

void main( )
    {
      int x;
      for(x=1; x<=10; x++)
      {
        cout<< x <<endl;
      }
      getch();
}
```

IV. 3. La boucle do. . . while

Dans certaines situations, il est nécessaire d'exécuter le corps de la boucle avant de tester la condition. Ces situations peuvent être traitées avec la boucle do. . . while. Do permet d'évaluer le corps de la boucle en premier, la condition est vérifiée en utilisant l'instruction while. Le format général de la boucle do. . . while est,

```
do
{
 ....
 .....
```

```
}
while(condition)
```

Exemple : programme pour afficher les dix premiers nombres multiplies par 5.

```
#include <iostream>
using namespace std;

void main()
{
      int a,i;
      a=5;
      i=1;
      do
      {
        cout << a*i <<endl;
        i++;
      }
      while(i <= 10);
     getch();
}
```

IV. 4. Les boucles imbriquées

Nous pouvons également avoir des boucles imbriquées, c'est-à-dire une boucle à l'intérieur d'une autre boucle. La syntaxe de base est :

```
for(initialization; condition; increment/decrement)
{
  for(initialization; condition; increment/decrement)
     {
        statement ;
     }
}
```

Exemple : Programme qui affiche une demi-pyramide de nombres

```
#include <iostream>
using namespace std;

void main( )
{
      int i,j;
      for(i=1;i<5;i++)
      {
        cout << "n";
        for(j=i;j>0;j--)
        {
          cout << j <<endl;
        }
      }
      getch();
}
```

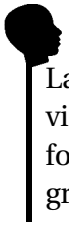
IV. 5. La boucle infinie

Une boucle devient une boucle infinie si une condition ne devient jamais fausse, elle est toujours vraie. ou vous pouvez faire une boucle infinie en laissant l'expression conditionnelle vide.

```
#include <iostream>
using namespace std;

int main ()
{
        for( ; ; ) {
           cout << "Cette une boucle infinie" <<endl;
        }
        return 0;
}
```

Lorsque l'expression conditionnelle est absente, elle est supposée vraie. Vous pouvez avoir une expression d'initialisation et d'incrémentation, mais les programmeurs $C++$ utilisent plus souvent la syntaxe $for(;;)$ pour signifier une boucle infinie. Vous pouvez terminer une boucle infinie en appuyant sur la touche $Ctrl+C$.



À retenir

La différence entre les deux boucles for et while réside dans l'organisation, si vous deviez augmenter à 10, il serait beaucoup plus propre et plus lisible d'utiliser la boucle for, d'ailleurs, si le nombre d'itération dépend d'une variable existante dans votre programme, il serait plus propre de faire une boucle while.

CHAPITRE 4

Chaînes de caractères et Énumérations

Plan de ce chapitre

I. Introduction

Dans ce cours les points suivants sont abordés :

- Manipulation des chaînes de caractères,
- Les opérations de traitement des chaînes,
- Utilisation de la classe **string**
- Les types énumération,
- Les types personnalisés.

Le C++ utilise le jeu de caractères **ASCII** (American Standard Code for Information Interchange).

Une chaîne de caractères littérale est une suite de caractères entourés d'apostrophes doubles " ", (double quote).

En C++, les chaînes de caractères sont représentées soit :

- Suivant la convention du langage **C**,
- Comme étant des objets via la classe **string**.

II. Opérations sur les chaînes en C

Exemple :

```cpp
#include <iostream>

using namespace std;

int main()
{
    char majminchif[]="Bonjour 2017";
    int taillemaj  = sizeof(majminchif);
    cout << "Taille de la chaîne : " << taillemaj << endl;
    for (int i=0; i<taillemaj; ++i)
    {
        cout << "----------------------------------------" << endl;
        cout << majminchif[i] << "(en decimal)     -> " << int(
           majminchif[i]) << endl;
        cout << majminchif[i] << "(en octal)       -> " << oct <<
           int(majminchif[i]) << endl;
        cout << majminchif[i] << "(en hexadecimal) -> " << hex <<
           int(majminchif[i]) << endl;
    }
    cout << "Fin du programme" << endl;
    return 0;
}
```

Ce programme donne en sortie :

```
Taille de la chaine : 13
------------------------
A (en decimal)     -> 66
A (en octal)       -> 102
A (en hexadecimal) -> 42
------------------------
      .
      .
      .
------------------------
 (en decimal)     -> 0
 (en octal)       -> 0
 (en hexadecimal) -> 0
```

Ce résultat laisse poser plusieurs questions :

- Pourquoi la **taille** de la chaîne est **13**, alors qu'elle ne contient que **12** caractères ?
- Pourquoi la déclaration de **majminchif** est suivie de [] ?
- Quelle est le rôle du mot-clé **sizeof** ?
- A quoi servent les mots-clés **oct** et **hex** ?

Le C dispose des fonctions suivantes pour traiter les chaînes :

- **strcpy**(chaine1,chaine2) : permet de copier chaine2 dans chaine1,
- **strcmp**(chaine1,chaine2) : comparer chaine1 et chaine2,

— **strlen**(chaine1) : la longueur de la chaine1,

— **strcat**(chaine1,chaine2) : concaténer chaine1 et chaine2,

— **strchr**(chaine1,car1) : localiser car1 dans chaine1.

Pour utiliser ces fonctions, il faut obligatoirement inclure le fichier d'en-tête <string.h>.

```
#include <iostream>
#include <string.h>
using namespace std;

int main()
{
    char texte1[] = "Temperature du four";
    char texte2[] = " du train 3";
    char texte3[25];

    cout << "Texte1 + Texte2 : " << strcat(texte1,texte2) << endl;
    cout << "Texte3 :" << strcpy(texte3,texte1) << endl;
    cout << "Taille de la chaine texte1 : " << strlen(texte1)  <<
       endl;
    cout << "Quel texte commence à partir de la lettre 'f' :" <<
       strchr(texte1,'f') << endl;

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

III. Les chaînes en C++

Le C++ offre plus de fonctionnalités pour manipuler les chaînes de caractères.
Il existe des fonctionnalités comme la recherche d'une chaine ou d'un caractère, la présence ou l'absence d'un caractère dans une suite, l'insertion, la suppression et le remplacement.

Exemple :

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string texte1, texte3;
    texte1 = "Temperature du four";


    texte1 += " du train 3";
    texte3 = texte1;

    cout << "texte1 : " << texte1 << endl;
```

```
15      cout << "texte3 : " << texte3 << endl;
16
17
18      cout << "Fin du programme" << endl;
19      return 0;
20  }
```

Ces fonctionnalités sont résumées dans le tableau suivant :

Fonction	**Rôle**
+	Concaténation
=	Affectation
begin()	Itérateur renvoyant le début
end()	Itérateur renvoyant la fin
size()	renvoie le nombre de caractères utiles d'une chaîne
at(int i)	récupérer le i-ème caractère. (0 = 1èr)
insert(position de départ, chaine)	Insertion dans une chaine
erase(position de départ, nb caractères à effacer)	Efface les caractères d'une chaîne
replace(position de départ, longueur, chaine)	Remplace les caractères d'une chaîne
find(chaine)	Cherche les caractères dans une chaine
find_first_of(chaine)	Cherche la première occurrence d'un caractère d'une chaîne dans une chaîne
find_last_of(chaine)	Cherche la dernière occurrence d'un caractère d'une chaîne dans une chaîne

TABLE 4.1 – Fonctionnalités de manipulation des chaines en C++

Exemple d'utilisation de ces fonctionnalités :

```
1   #include <iostream>
2   #include <string>
3
4   using namespace std;
5   int main (void)
6   {
7       string s = "BONJOUR";
8       int i, taille = s.size();
9
10      cout << "La chaine comporte " << taille << " caracteres." <<
            endl;
11
12      for(i = 0; i <taille ; i++)
13          cout << "caractere " << i << " = " << s.at(i) << endl;
14
15      return 0;
16  }
17      cout << "Fin du programme" << endl;
```

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

IV. Les types énumération

C++ offre la possibilité de définir ses propres types. Un *type énumération* est constitué d'un ensemble fini de valeurs appelées **énumérateurs**. Ils offrent une meilleure lisibilité du code.

Exemple :

```
1  #include <iostream>
2
3  using namespace std;
4
5  enum jour{lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche
       };
6  enum couleur{brun, cyan, rouge, vert, bleu, magenta, jaune, noir};
7  enum logique{vrai=1, faux=0};
8
9  int main()
10 {
11     // Déclaration
12     jour j1, j2;
13     j1 = mardi;
14     j2 = mardi;
15     couleur tomate = rouge;
16     couleur ciel = bleu;
17     logique ok = vrai;
18
19     if (j1==j2) cout << "Jours identiques" << endl;
20     if (tomate == 2) cout << "Une tomate est rouge" << endl;
21     if (ciel == 5) cout << "Un canari est jaune";
22     else cout << "Pas toujours" << endl;
23     cout << "Valeur de ok : " << ok << endl;
24
25     return 0;
26 }
```

Le résultat de ce programme est :

```
Jours identiques
Une tomate est rouge
Pas toujours
Valeur de ok : 1
```

Attention

Le carré du sinus d'un nombre n'est pas égal au sinus du carré du nombre.

CHAPITRE 5

Entrées - Sorties

Plan de ce chapitre

I. La classe istream

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

Les flots en entrée sont décrits par la classe **istream**.
L'objet **cin** est une instance de cette classe, automatiquement créé et destiné aux entrées depuis le clavier.
En plus de l'opérateur de lecture ≫ que nous avons déja utilisé, la classe **istream** dispose de nombreuses fonctions, dont les suivantes :

— **get()** 1^{ere} forme, déclarée ainsi :

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

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

Exemple :

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

— get() $2^{ème}$ forme, déclarée ainsi :

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

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

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

— get() $3^{ème}$ forme, déclarée ainsi :

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

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

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

— **getline()** déclarée ainsi :

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

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

II. Les fichiers textes

II. 1. Utilisation de fstream

II. 1. a. La classe ofstream :

Il s'agit d'un fichier ouvert en écriture : pour créer un tel fichier il suffit d'appeler le constructeur qui a en paramètre le nom du fichier :

```
ofstream f("toto.txt");
```

Pour savoir si le fichier a bien été ouvert en écriture la méthode **is_ open()** renvoie true si le fichier est effectivement ouvert.
Pour écrire dans le fichier on utilise l'opérateur ≪ sans oublier d'écrire des séparateurs dans le fichier texte.

II. 2. La classe ifstream :

Il s'agit d'un fichier ouvert en lecture : pour créer un tel fichier il suffit d'appeler le constructeur qui a en paramètre le nom du fichier :

```
ifstream f("toto.txt");
```

Pour savoir si le fichier a bien été ouvert en lecture la méthode **is_open()** renvoie true si le fichier est effectivement ouvert.
Pour lire dans le fichier on utilise l'opérateur ≫.

Exemple : écriture d'un fichier text

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

using namespace std;

int main(void)
 {
   int a = 78, t1 [6], i;
   double b = 9.87;
   char c = 'W';
   string s;

   for (i = 0; i < 6; i++)
    t1 [i] = 10000+i;
   s = "AZERTYUIO";
   ofstream f ("toto.txt");

   if (!f.is_open())
    cout << "Impossible d'ouvrir le fichier en écriture !" << endl;
   else
    {
     f << a << " " << b << " " << c << endl;
     for (i = 0; i < 6; i++)
       f << t1 [i] << " ";
     f << s;
    }
   f.close();
  return 0;
}
```

Exemple : lecture d'un fichier texte

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

using namespace std;

int main (void)
 {
   int a, t1 [6], i;
   double b;
   char c;
   string s;
   ifstream f ("toto.txt");

```

```
15    if (!f.is_open())
16      cout << "Impossible d'ouvrir le fichier en lecture !" << endl;
17    else
18     {
19      f >> a >> b >> c;
20      for (i = 0; i < 6; i++)
21        f >> t1 [i];
22      f >> s;
23     }
24    f.close();
25    cout << "a=" << a << endl
26     << "b=" << b <<endl
27     << "c=" << c <<endl;
28    for (i = 0; i < 6; i++)
29      cout << t1 [i] << endl;
30    cout << s << endl;
31
32    return 0;
33 }
```

III. Les fichiers binaires

III. 1. La classe ofstream

Pour ouvrir en écriture un fichier binaire, il suffit d'appeler le constructeur qui a en paramètre le nom du fichier et le mode d'ouverture et fixer ce deuxième paramètre à $ios::out|ios::binary$:

```
ofstream f("file.abc",ios::out | ios::binary);
```

Pour savoir si le fichier a bien été ouvert en écriture la méthode **is_ open()** renvoie true si le fichier est effectivement ouvert.
Pour écrire dans le fichier on utilise la méthode $write((char*)buffer,intnb)$ pour écrire **nb** octets dans ce fichier.

III. 2. La classe ifstream

Pour ouvrir en lecture un fichier binaire, il suffit d'appeler le constructeur qui a en paramètre le nom du fichier et le mode d'ouverture et fixer ce deuxième paramètre à $ios::in|ios::binary$

```
ifstream f("file.abc",ios::in | ios::binary);
```

Pour savoir si le fichier a bien été ouvert en écriture la méthode **is _open()** renvoie true si le fichier est effectivement ouvert.
Pour lire dans le fichier on utilise la méthode $read((char*)buffer,intnb)$ pour lire **nb** octets de ce fichier.

Exemple : Écriture d'un fichier binaire

```cpp
#include <iostream>
#include <fstream>

using namespace std;

int main (void)
{
   int a = 78, t1 [6], i;
   double b = 9.87;
   char c = 'W';

   for (i = 0; i < 6; i++)
     t1 [i] = 10000+i;

   ofstream f ("toto.xyz", ios::out | ios::binary);

   if(!f.is_open())
     cout << "Impossible d'ouvrir le fichier en écriture !" << endl
        ;
   else
    {
     f.write ((char *)&a, sizeof(int));
     f.write ((char *)&b, sizeof(double));
     f.write ((char *)&c, sizeof(char));
     for (i = 0; i < 6; i++)
       f.write ((char *)&t1[i], sizeof(int));
    }
    f.close();

  return 0;
}
```

Exemple : lecture d'un fichier binaire

```cpp
#include <iostream>
#include<fstream>
using namespace std;

int main (void)
{
   int a, t1 [6], i;
   double b;
   char c;
   string s;
   ifstream f ("toto.xyz", ios::in | ios::binary);

   if (!f.is_open())
     cout << "Impossible d'ouvrir le fichier en lecture !" << endl;
   else
    {
     f.read ((char *)&a, sizeof(int));
     f.read ((char *)&b, sizeof(double));
     f.read ((char *)&c, sizeof(char));
     for (i = 0; i < 6; i++)
```

```
21      f.read ((char *)&t1[i], sizeof(int));
22     }
23     f.close();
24
25    cout << "a=" << a << endl
26     << "b=" << b << endl
27     << "c=" << c << endl;
28    for (i = 0; i < 6; i++)
29      cout << t1 [i] << endl;
30
31   return 0;
32 }
```

CHAPITRE 6

Pointeurs et Références

Plan de ce chapitre

I. Introduction

Pour créer un lien vers une information (une donnée), on utilise des références ou des pointeurs.

Les pointeurs servent essentiellement à trois choses :

1. à permettre à plusieurs portions de code de partager des objets (données, fonctions, ...) sans les dupliquer, on parle de **références** ;
2. à pouvoir choisir des éléments non connus à priori (au moment de la programmation), on parle de la **généricité** ;
3. à pouvoir manipuler des objets dont la durée de vie dépasse la portée, on parle alors de l'**allocation dynamique**.

II. Les références

II. 1. Définitions

— Une *référence* est un autre nom pour un objet existant (un synonyme, un alias).

— Une *référence* permet de désigner un objet indirectement.

La déclaration se fait de la manière suivante :

```
type& nom_reference(identificateur);
```

Exemple :

```
int val(1);
int& x(val);
```

Attention :

1. Une référence est une étiquette et **PAS UNE VARIABLE!**

```
int i(3);
int& j(i);
i=4; /* j vaut 4 */
j=6; /* i vaut 6 */
```

```
int i(3);
int j(i);
i=4; /* j vaut 3 */
j=6; /* i vaut 4 */
```

2. Attention à la sémantique de **const** :

```
int i(3);
const int& j(i);
j=12; // NON
i=12; // OUI, et j vaut 12!
```

II. 2. Spécificités des références

1. Une référence **DOIT ABSOLUMENT ÊTRE INITIALISÉE** vers un objet existant :

```
int i;
int& ri(i); // OUI
int& rj;    // NON
```

2. Une référence **NE PEUT ÊTRE LIÉE QU'A UN SEUL OBJET** :

```
int i;
int& ri(i); // ri : référence vers i
int j(2);
ri = j;  // On ne peur pas déplacer la référence (étiquette) vers j
j = 3;
cout << i << endl; // Affiche 2
```

3. On ne peut pas **RÉFÉRENCER UNE RÉFÉRENCE** :

```
int i(3);
int& ri(i); // ri : référence vers i
int& rri(ri);    // NON
int&& rri(ri);   // NON PLUS
```

III. Les pointeurs

√ La principale différence c'est que le pointeur est une variable et non pas une étiquette vers une variable.

√ Une variable est identifiée par son adresse en mémoire (l'emplacement mémoire qui contient sa valeur).

√ un pointeur est une variable qui contient l'adresse d'un autre objet (une variable de variable).

√ Déclaration :

```
int* ptr;
```

√ Affectation :

```
ptr = &x;
```

√ Allocation :

```
ptr = new int(123);
```

Attention : Le pointeur n'est pas égal à la valeur pointée.

√ Libération :

```
delete ptr;
```

√ Copier un pointeur :

```
p1 = p2; // echanger des adresses
```

√ Annuler(Effacer) un pointeur :

```
p1 = nullptr;
```

IV. Déclaration et opérateurs de base

1. La déclaration d'un pointeur se fait par :

   ```
   type* identificateur;
   ```

   Exemple :

   ```
   int* ptr;
   double* ptr1;
   ```

2. L'initialisation d'un pointeur se fait par :

   ```
   type* identificateur(adresse);
   ```

   Exemple :

```
int* ptr(nullptr);
int* ptr(&i)
int* ptr(new int(33));
```

3. L'opérateur [&] : retourne l'adresse mémoire de la valeur d'une variable.
   Si x est de type **type**, **&x** est de type **type**$*$ (pointeur sur type).
   Exemple :

   ```
   int x(3);
   int* px(nullptr);
   px = &x;
   ```

4. L'opérateur [$*$] : retourne la valeur pointée par une variable pointeur.
   Exemple :

   ```
   int x(3);
   int* px(nullptr);
   px = &x;
   cout << *px << endl;
   ```

V. Allocation dynamique

Il y a deux façons d'allouer de la mémoire en C++.

1. Allocation **statique** qui se fait pendant la phase de compilation (avant l'exécution).
   Exemple :

   ```
   int v;
   ```

2. Allocation **dynamique** (au cours de l'exécution du programme).
   Exemple :

   ```
   vector<int> v;
   v.push_back(3);
   ```

Deux opérateurs existent pour ce fait :

— new : allocation mémoire

— delete : libération mémoire

Syntaxe :

```
pointeur = new type;
ex: int* px;

pointeur = new type(valeur);
ex: px = new int(3);

delete pointeur;
ex: delete px;
    px = nullptr;
```

Exemple :

```
1 int* px(nullptr);
2 px = new int;
3 *px  = 20;
4 cout << *px << endl;
5 delete px;
6 px = nullptr;
```

Attention à ceci :

```
int* px;
*px  = 20;
cout << *px << endl;

Ceci produit l'erreur: Segmentation Fault.
Solution : int* px(new int).

Il est conseillé d'initialiser le pointeur par : int* px(nullptr);
```

CHAPITRE 7

Vecteurs et Tableaux

Plan de ce chapitre

Un **tableau** est une suite de données homogènes(de même type).

En C++, chaque élément du tableau va être repéré via un **indice**.

Par l'intermédiaire des indices qui déterminent la position d'une cellule dans le tableau nous aurons un **accès direct** à la donnée contenue.

La suite des indices démarre toujours à partir de **0**, et s'incrémente par pas de **1**.

Il existe quatre(04) types de tableaux en C++ :

		taille initiale connue à priori?	
		Non	**Oui**
taille pouvant varier lors de l'utilisation du tableau	**Oui**	1. vector	2. vector
	Non	3. vector	4. array (C++11)

Exemple : Un tableau *orientation* contenant trois(03) float.

+20.0	-38.0	+137.0
orientation[0]	orientation[1]	orientation[2]

I. Les tableaux dynamiques

Un tableau dynamique est une collection de données homogènes dont le **nombre** peut **changer** au cours du déroulement du programme (ajout, retrait d'éléments).

Les tableaux dynamiques en C++ sont définit par le type **vector**. Pour les utiliser, il faut tout d'abord inclure :

```
#include <vector>
```

I. 1. Déclaration

On utilise la syntaxe suivante :

```
vector<type> identificateur;
```

Exemple :

```
#include <vector>
...
vector<int> tableau;
```

I. 2. Initialisation

Il existe cinq(05) façons d'initialiser un tableau dynamique.

1. **Tableau vide** (sans élément) :
   Exemple :

   ```
   vector<int> tableau;
   vector<double> mesures;
   ```

2. Avec des **valeurs initiales** différentes :

   ```
   vector<type> identificateur({val1, val2,..., valn});
   ```

   Exemple :

   ```
   vector<int> ages({20,30,40,50,60});
   ou
   vector<int> ages = {20,30,40,50,60};
   ```

3. Une taille initiale peut-être indiquée si nécessaire :

   ```
   vector<type> identificateur(taille);
   ```

   Exemple :

```
vector<int> tab(5); // un tableau de 5 entiers tous nuls
```

4. Avec taille initiale et initialisation des éléments à la même valeurs.

   ```
   vector<type> identificateur(taille, valeur);
   ```

   Exemple :

   ```
   vector<int> tab(5,1);
   ```

5. Initialiser un tableau dynamique à l'aide d'un **copie d'un autre tableau dynamique**.

   ```
   vector<type> identificateur(référence);
   ```

   ou *référence* est une référence à un tableau de même type.

   ```
   vector<int> tab1(5, 1);
   vector<int> tab2(tab1);
   ```

II. Utilisation des tableaux dynamiques

II. 1. Affectation globale

```
tableau1 = tableau2;
```

ou *tableau1* et *tableau2* sont deux tableaux de même type.
Exemple :

```
#include <vector>
...
vector<int> tab1({1,2,3,4});
vector<int> tab2;
...
tab2 = tab1;
```

II. 2. Accès direct aux éléments d'un tableau

Le $(i+1)^{ième}$ élément d'un tableau tab est référencé par :

```
tab[i]
```

Exemple : Pour un tableau a 5 éléments.

```
Premier élément tab[0];
Dernier élément tab[4];
```

Attention à la manipulation suivante :

```
vector<double> v;
v[0] = 4.0; // v[0] n'existe pas encore.
```

II. 3. Accès par itération au tableau

Il existe deux façons :

II. 3. a. Itération sur ensemble de valeurs

Pour parcourir l'ensemble des éléments d'un tableau en C++, on utilise la boucle suivante :

1. Si l'on veut pas modifier les éléments du tableau :

   ```
   for(auto nom_de_variable : tableau)
   ```

   ou *nom_de_variable* est une variable utilisée pour parcourir tous les éléments du tableau.

2. Si l'on veut modifier les éléments du tableau :

   ```
   for(auto& nom_de_variable : tableau)
   ```

Exemple : Itération sur l'ensemble des éléments

```
#include <vector>
...
vector<int> ages(5);
for(auto& age : tableau)
  {
    cout << "Age de l'employe suivant : ";
    cin >> age;
  }
cout << "Age des employes : " << endl;
for(auto age : tableau)
  {
    cout << "    " << age << endl;
  }
...
```

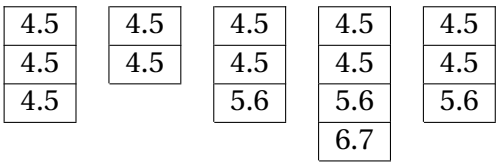
II. 3. b. Itération utilisant la boucle *for* classique

Dans ce cas il faut connaitre au préalable la taille du tableau. Ceci est fait par la fonction *size()*.
Cette taille est de type *size_t*.

```
for (size_t i(0); i<tab.size(); ++i)
```

Exemple :

```
#include <vector>
...
vector<int> ages(5);
for(size_t i(0); i<ages.size(); ++i)
  {
    cout << "Age de l'employe : " << i+1 << " ? ";
    cin >> ages[i];
  }
...
```



III. Fonctions spécifiques pour *vector*

Ces fonctions s'écrivent suivant la syntaxe :

```
nom_de_tableau.nom_de_fonction(arg1, arg2, ..., argn)
```

Exemple :

```
#include <vector>
...
vector<double> mesures(5);
size_t nombre_mesures(0);
...
nombre_mesures = mesures.size();
...
```

Il existe d'autres fonctions :

1. **tableau.front()** : donne le premier élément d'un tableau ou $tableau[0]$;
2. **tableau.back()** : donne le dernier élément d'un tableau ou $tableau[tableau.size() - 1]$;
3. **tableau.empty()** : détermine si un tableau est vide (résultat : true ou false) ;
4. **tableau.clear()** : supprime tous les éléments d'un tableau ;
5. **tableau.pop_back()** : supprime le dernier élément d'un tableau ;
6. **tableau.push_back(val)** : ajoute un élément de valeur *val* à la fin d'un tableau ;

Exemple :

```
#include <vector>
vector<int> v(3, 4.5);
...
v.pop_back();
v.push_back(5.6);
v.push_back(6.7);
v.pop_back();
...
```

IV. Tableaux dynamiques multidimensionnels

Pour déclarer un tableau a plusieurs dimensions, il suffit d'ajouter un niveau de plus(c'est un tableau de tableaux). *Exemple* :

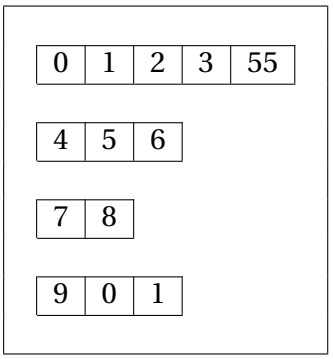
```
vector<vector<int> > tab(5, vector<int>(6));
```

correspond à la déclaration d'un tableau de 5 tableaux de 6 entiers.
Avec :
tab[i] est un vecteur *vector<int>* (tableau dynamique d'entiers).
tab[i][j] sera alors le $(j+1)^{ieme}$ élément de ce tableau.

Attention : C'est un tableau dynamique de tableaux dynamiques (qui n'ont pas forcément la même taille !).

Exemple :

```
#include <vector>
...
vector<vector<int> > tableau({
{0,1,2,3,55},
{4,5,6},
{7,8},
{9,1,0}
});
...
for (auto ligne : tableau) {
  for (auto element : ligne) {
    cout << element << "  ";
    }
  cout << endl;
  }
for(size_t i(0); i<tableau.size(); ++i) {
  cout << "tableau [" << i << "].size() = " << tableau[i].size() <<
      endl;
  }
```



V. Les Array

Une **array** (ou tableau statique ou tableau de taille fixe) est un tableau dont la taille initiale est connue à priori et ne pouvant pas varier.

V. 1. Utilisation

Pour utiliser les array, il faut d'abord inclure l'en-tête :

```
#include <array>
```

V. 2. Déclaration

On déclare une array en utilisant la syntaxe suivante :

```
#include <array>
...
array<type, taille> identificateur;
...
```

Exemple 1 :

```
array<double, 3> vecteur_3D;
```

Exemple 2 :

```
const size_t nb_wilayas(48);
array<unsigned int, nb_wilayas> habitants;
```

Exemple 3 :

```
const size_t taille(5);
array<int, taille> ages({20,30,40,50});
array<int, taille> ages = {20,30,40,50};
```

Exemple 4 :

```
array<array<int, 3>, 4>
matrice = {
      0,1,2,
      3,4,5,
      6,7,8,
      9,0,1
      };
```

0	1	2
3	4	5
6	7	8
9	0	1

VI. Construction d'un tableau suivant C

Tous les compilateurs C++ ne supportent pas l'utilisation des classes *vector* et *array*, il est donc intéressant de connaitre l'écriture du code suivant la syntaxe **C**.

Exemple 1 : L'utilisation d'un *vecteur*.
La déclaration est identique à celle d'une variable ordinaire que l'on fait suivre de son **indice** entre crochets.

```cpp
#include <iostream>
using namespace std;
int main() {

  float note[6];
  float somme = 0;
  int i;

  for(i=0; i<6; ++i)
  {
    cout << "Entrez la note N# " << i+1 << " : ";
    cin >> note[i];
  }

  for(i=0; i<6; ++i)
    somme += note[i];
    cout << "Moyenne = " << somme/6 << endl;

    return 0;
}
```

Exemple 2 : Une matrice (tableau multi-dimensionnel).

```cpp
#include <iostream>
using namespace std;
int main() {

  float pt[3][2];
  float poids =0, taille = 0;
  int i,j;

  for(i=0; i<3; i++)
  {
    for(j=0; j<2; j++)
    {
      cout << " Indices : [" << i << "][" << j << "]"  << endl;
      if (j == 0) {
              cout << "Poids : ";
              cin >> pt[i][j];
            }
      else {
           cout << " Taille : ";
           cin >> pt[i][j];
          }
    }
  }
  for (i=0; i<3; i++)
    for (j=0; j<3; j++)
    {
      if (j == 0) poids += pt[i][j];
      else taille += pt[i][j];
    }
    cout << "Moyenne poids = " << poids/3 << endl;
    cout << "Moyenne taille = " << taille/3 << endl;
```

```
32
33      return 0;
34  }
```

En C++, on peut initialiser un tableau via une liste d'initialisation . les valeurs de la liste sont attribuées à chacun des éléments du tableau en respectant leur ordre d'apparition.

L'exemple suivant montre la syntaxe à utiliser. Le nombre d'éléments de la liste détermine la taille du tableau si celle-ci n'a pas été précisée.

```
1  #include <iostream>
2  using namespace std;
3  int main() {
4
5    int carre[2][3] ={{1,2,3},{1,4,9}};
6
7    for(int ligne=0; ligne<2; ligne++)
8    {
9      for(int colonne=0; colonne<3; colonne++)
10         cout << carre[ligne][colonne] << endl;
11     }      return 0;    }
```

CHAPITRE 8

Les Fonctions

Plan de ce chapitre

Dans ce cours les points suivants sont abordés :

— Syntaxe et utilisation,
— Récursivité,
— Passage par référence,
— Surcharge d'une fonction.

Les fonctions offrent au programmeur la possibilité de décomposer un ensemble de code complexe en une série de sous-ensembles plus petits, assurant une meilleure lisibilité , une modularité et une simplification de la mise au point.

I. Syntaxe et utilisation

Une fonction est bâtie autour d'un ensemble d'instructions assurant une tâche particulière.
Une fonction doit-être déclarée (**prototype** de la fonction) avant d'être utilisée.
L'**implémentation** d'une fonction c'est son contenu en instructions.

La syntaxe générale est la suivante :

```
type nomFonction(arg1, arg2, ...argn);
.
.
.
int main()
{
  ....
  // appel de la fonction nomFonction
  nomFonction(p1, p2, ...,pn);
```

```
  ....
  return 0;
}

type nomFonction(arg1, arg2, ...argn)
{
  // To Do
  return expression;
}
```

Exemple :

```
#include <iostream>
#include <cmath>
using namespace std;

int nbpremier(int);

int main()
{
  int x;
  cout << "Saisissez un nombre entier : ";
  cin >> x;
  if(nbpremier(x) == 0) cout << "Le nombre " << x << " n'est pas un
      nombre premier" <<
  endl;
  else
  cout << "Le nombre est premier" << endl;
  return 0;
}

int nbpremier(int n)
{
  int d;
  float racine_n = sqrt(n);
  if (n < 2) return 0;
  if (n ==2) return 1;
  if (n % 2 ==0) return 0;
  for(d=3; d<=racine_n; d+=2)
    if(n % d == 0) return 0;
    return 1;
}
```

Le résultat de ce programme est le suivant :

```
Saisissez un nombre entier : 87
Le nombre 87 n'est pas un nombre premier
      .
Saisissez un nombre entier : 15773
Le nombre est premier
```

Une deuxième alternative :

```
#include <iostream>
using namespace std;
```

```
3
4  float puissance()
5  {
6    float nb, resultat = 1.0;
7    int exp;
8    cout << "Saisissez un nombre : ";
9    cin >> nb;
10   cout << "Saisissez un exposant : ";
11   cin >> exp;
12   for (int i=1; i<=exp; ++i) resultat *= nb;
13   return resultat;
14 }
15
16 int main()
17 {
18   cout << "Resultat : " << puissance() << endl;
19   return 0;
20 }
```

II. La récursivité

Une fonction peut s'appeler d'elle même, ceci est appelé **récursion** ou **récursivité** : des calculs répétitifs jusqu'à ce qu'une condition soit vérifiée.

Exemple :

```
1  #include <iostream>
2  using namespace std;
3
4  int factoriel(int);
5
6  int main()
7  {
8    int nombre;
9    cout << "Saisissez un nombre entier : ";
10   cin >> nombre;
11   cout << nombre << "! = " << factoriel(nombre) << endl;
12   return 0;
13 }
14 int factoriel(int n)
15 {
16   if (n == 1) return n;
17   else
18   n = n * factoriel(n-1);
19   return n;
20 }
```

III. Passage par référence et renvoi multiples

III. 1. Passage par référence

Jusqu'à présent, nous avons utilisé le passage des arguments vers la fonction **par valeur**.
Pour passer un *argument par référence*, le *type* spécifié de l'argument est suivi de **&**. La *variable locale* devient alors une référence à l'*argument effectif* qui lui ait passé. Cet *argument effectif* est maintenant en **lecture-écriture**.

Exemple :

```
#include <iostream>
using namespace std;

void test(int x, int& y)
{
  x = 100;
  y = 1000;
}

int main()
{
  int i = 99, j = 999;
  cout << "i = " << i << " et j = " << j << endl;
  test(i,j);
  cout << "i = " << i << " et j = " << j << endl;
  return 0;
}
```

Ce qui donne comme résultat :

```
i = 99 et j = 999
i = 99 et j = 1000
```

III. 2. Renvoi multiples

Les références permettent à une fonction de renvoyer plusieurs résultats au d'un seul.
Exemple :

```
#include <iostream>
using namespace std;

void sphere(double&, double&, double&, double);

int main()
{
  double r, v, s, c;
  cout << "Saisissez le rayon : ";
  cin >> r;
  sphere(v, s, c, r);
  cout << "Volume         = " << v << endl;
```

```
13    cout << "Surface       = " << s << endl;
14    cout << "Circonférence = " << c << endl;
15    return 0;
16  }
17
18  void sphere(double& volume, double& surface, double& circonference,
        double rayon)
19  {
20    const double PI = 3.14159265;
21    volume = 4.0/3.0*PI*rayon*rayon*rayon;
22    surface = 4*PI*rayon*rayon;
23    circonference = 2*PI*rayon;
24  }
```

IV. Surcharge d'une fonction

La surchage permet l'utilisation de fonctions de même noms qui portent des arguments différents.
Exemple :

```
1   #include <iostream>
2   using namespace std;
3
4   double volume(double);
5   double volume(double, double);
6   double volume(double, double, double);
7
8   int main()
9   {
10    int choix;
11    double cote, rayon, hauteur, longueur, largeur, haut;
12    cout << "Saisissez le volume à calculer (1-Cube, 2-Cône, 3-Paralé
        lépipède : ";
13    cin >> choix;
14
15    switch(choix)
16    {
17      case 1:
18        cout << "Cote du cube";
19        cin >> cote;
20        cout << "Le volume du cube est égal à : " << volume(cote) <<
            endl;
21        break;
22      case 2:
23        cout << "Rayon du cône : ";
24        cin >> rayon;
25        cout << "Hauteur du cône";
26        cin >> hauteur;
27        cout << "Le volume du Cône est égal à : " << volume(rayon,
            hauteur) << endl;
28        break;
29      case 3:
```

```
30        cout << "Longueur du Paralélépipède";
31        cin >> longueur;
32        cout << "Largeur du Paralélépipède";
33        cin >> largeur;
34        cout << "Hauteur du Paralélépipède";
35        cin >> haut;
36        cout << "Le volume du Paralélépipède est égal à : " << volume
             (longueur, largeur,
37        haut) << endl;
38        break;
39      default:
40        cout << "Erreur !!!" << endl;
41        break;
42    }
43  }
44
45  double volume(double c)
46  {
47    double v;
48    v = c*c*c;
49    return v;
50  }
51
52  double volume(double r, double h1)
53  {
54    double v;
55    const double PI = 3.14159265;
56    v = PI/3.0*r*r*h1;
57    return v;
58  }
59
60  double volume(double L, double l, double h2)
61  {
62    double v;
63    v = L*l*h2;
64    return v;
65  }
```

CHAPITRE 9

Programmation orientée objet

Plan de ce chapitre

I. Introduction

La programmation orientée objet est un moyen pour résoudre des problèmes complexes en les décomposant en problèmes plus petits à l'aide des objets. Avant la programmation orientée objet (POO), les programmes étaient écrits en langage procédural, ils ne constituaient qu'une longue liste d'instructions.

En programmation orientée objet, nous écrivons des programmes utilisant des classes, des objets, des fonctionnalités telles que l'abstraction, l'encapsulation, l'héritage et le polymorphisme.

II. Concepts de classes et objets

Un Objet : Toute entité ayant un état et un comportement s'appelle un objet. Par exemple : voiture, maison, chaise, stylo, clavier, vélo, etc. Cela peut être physique et logique. On peut voir un objet comme une boite (figure 9.1) qui contient les données qui le caractérise (C'est ce que l'on appelle l'encapsulation) et les méthodes qui permettent de manipuler ces données.
Exemple :

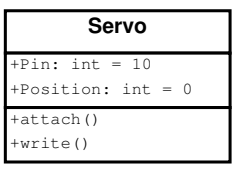


FIGURE 9.1 – Représentation d'une **classe** Servo

Une Classe : Une collection d'objets s'appelle "classe". C'est une entité logique.

Exemple :

```
class Servo
{
public:
  Servo();
  uint8_t attach(int pin);
private:
   uint8_t servoIndex;
   int8_t min;
   int8_t max;
};
```

Un exemple d'utilisation :

```
Servo serrure;
serrure.attach(10);
serrure.write(90);
```

Servo est un **objet**, pour lequel deux **fonctions membres** existent : **attach** et **write**. **Servo** *serrure* : crée un objet nommé *serrure* de type **Servo**.

III. Visibilité et Encapsulation

Pour bien comprendre ces notions, nous allons développer une **classe** qui va nous servir d'exemple. Soit une **LED**, qui doit être initialisée avec un numéro de broche (d'un microcontrôleur) et qui peut s'allumer et s'éteindre, donc avec une variable qui stockera son état.

La déclaration pourrait ressembler à ça :

```
    class Led
    {
      // Données
      byte pin;
      byte etat;
      // Méthodes
      void Setup(byte aPin);  // initialisation de la broche pour
         la led.
      void Allumer();
      void Eteindre();
    };
```

À retenir

Le "**;**" est important à la fin de la définition de la classe. Il est **obligatoire**.

L'encapsulation consiste à regrouper sous un même nom tout ce qui concerne un type d'objet, données et méthodes. Elle permet également de ne laisser accessible que ce qui doit l'être vu de l'extérieur. Il y a trois niveau de protection du contenu d'une classe : **privé**, **protégé** et **public**.

— Si rien n'est précisé comme dans notre exemple LED, tout est privé (**private**). Ce qui est privé n'est visible que des objets de la classe elle même.

— Ce qui est déclaré **protected** (protégé) est visible par la classe elle même, mais aussi par les classes dérivées (héritage).

— Les éléments publics sont visibles par tous.

À retenir

Cette version de la classe LED ne peut pas fonctionner. Tout est privé, ce qui empêche l'accès aux méthodes Setup(), Allumer() et Eteindre().

Une autre version plus ouverte de la classe :

```
class Led
{
private:
  byte pin;
  byte etat;

public:
  void Setup(byte aPin);
  void Allumer();
  void Eteindre();
};
```

Attention : cet exemple d'utilisation est incorrect

```
int main()
{
Led danger;
danger.pin = 10; // Erreur de compilation, pin n'est pas accessible.
}
```

Ajoutons plus de fonctionnalités. Pour une LED, selon la façon dont elle est câblée, un état haut (**HIGH**) de la broche peut l'allumer ou l'éteindre.

Pour que la classe LED fonctionne dans les deux cas, il faut définir le type de montage de la LED : **normal** (broche reliée a l'anode et fournissant le courant) ou **inversé** (broche reliée a la cathode et absorbant le courant), afin d'être sur que lorsqu'on appelle la méthode **Allumer()**, le résultat est bien celui espéré. Pour simplifier le codage, passons alors par une fonction privée **Rafraichir** qui fera l'interprétation voulue.

```
class Led
{
private:
  byte pin;
  byte etat;  // HIGH pour allumé, LOW pour éteint,
              // quelque soit la valeur de montageInverse
  bool montageInverse; // true si il faut LOW pour allumer !
  void Rafraichir();

public:
  void Setup(byte aPin, bool aMontageInverse = false);
```

```
12    void Allumer();
13    void Eteindre();
14  };
15
16  // Définition des méthodes.
17  void Led::Setup(byte aPin, bool aMontageInverse)
18  {
19    montageInverse = aMontageInverse;
20    pin = aPin;
21    pinMode(aPin, OUTPUT);
22
23    // Commençons diode éteinte...
24    Eteindre();
25  }
26
27  void Led::Allumer()
28  {
29    etat = HIGH; // Allumé
30    Rafraichir();
31  }
32
33  void Led::Eteindre()
34  {
35    etat = LOW; // Eteint
36    Rafraichir();
37  }
38
39  void Led::Rafraichir()
40  {
41    byte vraiEtat = etat;          // on prend l'état demandé
42    if (montageInverse == true)    // si c'est un montage inversé
43      vraiEtat = ! vraiEtat;       // on inverse
44    digitalWrite(pin, vraiEtat);
45  }
46
47  // Programme C++
48
49  Led maled;
50
51  void Init()
52  {
53    maled.Setup(10, false);
54  }
```

Les méthodes **Allumer()** et **Eteindre()** se contentent de mettre à jour **etat**, puis la méthode privée **Rafraichir()** est appelée et s'occupe de modifier l'état matériel de la broche en fonction de l'état demandé et du type de montage.

À retenir

Noter la valeur par défaut de l'argument dans la déclaration de la méthode Setup :

```
void Setup(byte aPin, bool aMontageInverse = false);
```

Le fait de donner une valeur initiale à **aMontageInverse** permet d'omettre cet argument en appelant **Setup()** quand sa valeur est false. Cela permet d'écrire soit maled.Setup(10); et aMontageInverse est initialisé automatiquement à false, soit maled.Setup(10, false);, sinon il faut spécifier un état inverse avec maled.Setup(10, true);.

IV. Constructeurs/Destructeurs

A un moment donné, il faut créer l'objet, c'est à dire mettre en place en mémoire les variables de l'objet et les initialiser. On parle d'instanciation. Ainsi lorsque l'on écrit :

```
Led maled;
```

On fait appel implicitement à une fonction très particulière : le **constructeur**. C'est la première méthode de l'objet à être exécutée. La définition de cette méthode spéciale n'est pas obligatoire (si vous ne souhaitez pas initialiser les données par exemple) dans la mesure où une version par défaut de ce constructeur sans arguments existe toujours, automatiquement créée par le compilateur si elle n'est pas fournie par le programmeur, mais dans ce cas le contenu des données membres de l'objet créé est indéterminé. (Typiquement, les entiers ne contiendront pas forcément 0 après la création de l'objet, mais plutôt n'importe quoi.)

Cette méthode particulière est facilement identifiée par le fait que son nom est celui de la classe, et qu'elle n'accepte pas de valeur de retour.

Par symétrie avec le constructeur, il y a aussi un destructeur optionnel, dont le nom commence par un caractère ~.

```
1  class Led
2  {
3  private:
4    byte pin;
5    byte etat;  // HIGH pour allumé, LOW pour éteint,
6                // quelque soit la valeur de montageInverse
7    bool montageInverse; // true si il faut LOW pour allumer !
8    void Rafraichir();
9
10 public:
11   Led(byte aPin, bool aMontageInverse = false);  //
       constructeur complet
12   void Setup();
13   void Allumer();
14   void Eteindre();
15 };
16
17 // Définition des méthodes. Par convention, on commence par le
     constructeur
18 Led::Led(byte aPin, bool aMontageInverse)
```

```
19      {
20        pin = aPin;
21        montageInverse = aMontageInverse;
22      }
23
24      void Led::Setup()
25      {
26        pinMode(pin, OUTPUT);
27        Eteindre();
28      }
29
30      void Led::Allumer()
31      {
32        etat = HIGH; // Allumé
33        Rafraichir();
34      }
35
36      void Led::Eteindre()
37      {
38        etat = LOW; // Eteint
39        Rafraichir();
40      }
41
42      void Led::Rafraichir()
43      {
44        byte vraiEtat = etat;          // on prend l'état demandé
45        if (montageInverse == true)  // si c'est un montage inversé
46          vraiEtat = ! vraiEtat;     // on inverse
47        digitalWrite(pin, vraiEtat);
48      }
49      /// Fin de la classe Led
50
51
52      // On crée 2 leds
53      Led rouge(10);  // pin 10
54      Led verte(11);  // pin 11
55
56      void setup()
57      {
58        rouge.Setup();
59        verte.Setup();
60      }
61
62      void loop()
63      {
64        rouge.Allumer();
65        delay(1000);
66        rouge.Eteindre();
67        delay(1000);
68        verte.Allumer();
69        delay(1000);
70        verte.Eteindre();
71        delay(1000);
```

```
72      }
```

V. L'héritage

Si l'on veut s'inspirer de la classe LED pour produire une classe LEDBicouleur qui s'occuperait d'une LED à deux couleurs dotée de trois broches, l'héritage permet de dériver des objets à partir d'objets existants.
En voici un exemple :

```
1       class LedBicouleur : public Led
2       {
3       private
4         byte pin2;
5
6       public:
7         LedBicouleur(byte aPin1, byte aPin2, bool aMontageInverse =
              false);
8         void Allumer2();
9         void Allumer12();
10      };
11      LedBicouleur::LedBicouleur(byte aPin1, byte aPin2, bool
            aMontageInverse) : Led(aPin1, aMontageInverse)
12      {
13        pin2 = aPin2;
14        pinMode(pin2, OUTPUT);
15
16        Eteindre();
17      }
```

Cette led particulière ayant deux pins différentes à piloter, il faut en ajouter une à celle déjà présente dans la classe de base, c'est pin2, et l'initialiser dans le constructeur.

Ce constructeur ne va pas fonctionner correctement. Voyez vous ce qui ne va pas ? Eteindre() n'est présent que dans Led, et pas dans LedBicouleur, ce qui fait que cette méthode ne connait pas pin2 et donc n'éteindra pas cette broche... Nous devons faire en sorte qu'Eteindre() marche pour les deux classes.

Dans la nouvelle classe, on veut que etat = LOW éteindrait les deux broches. C'est donc Rafraichir() qui doit faire ce travail. Mais cette méthode existe déjà dans Led, comment en faire une nouvelle dans LedBicouleur qui mette à jour les deux broches ?

Pour cela, il y a la surcharge de méthode.

```
1       class Led
2       {
3       private:
4         byte pin;
5         byte etat;
6         bool montageInverse;
7         virtual void Rafraichir();
8
9       public:
```

```
10        Led(byte aPin, bool aMontageInverse = false);
11        void Allumer();
12        void Eteindre();
13      };
14
15      class LedBicouleur : public Led
16      {
17      private:
18        byte pin2;
19        void Rafraichir();
20
21      public:
22        LedBicouleur(byte aPin1, byte aPin2, bool aMontageInverse =
             false);
23        void Allumer2();
24      };
```

Dans la classe Led, est apparu le mot clé **virtual** qui déclare la fonction **Rafraichir()** virtuelle. Ce terme désigne une fonction qui dépend du type réel de la classe qui l'applique. Si c'est une Led, on utilisera Led : :Rafraichir(), si c'est une LebBicouleur on utilisera LedBicouleur : :Rafraichir(). En effet, virtual ouvre une porte et permet aux classes dérivées de fournir un comportement alternatif. Dans LedBicouleur, j'ai effectivement redéfini, surchargé Rafraichir().

Le code de **Rafraichir()** deviendra :

```
1     void LedBicouleur::Rafraichir()
2     {
3       byte etatLed1 = LOW;
4       byte etatLed2 = LOW;
5
6       switch(etat)
7       {
8         case HIGH:
9           etatLed1 = HIGH;
10          break;
11
12        case HIGH2:
13          etatLed2 = HIGH;
14          break;
15
16        case HIGH12:
17          etatLed1 = HIGH;
18          etatLed2 = HIGH;
19          break;
20      }
21
22      if (montageInverse == true)  // si c'est un montage inversé
23      {
24        etatLed1 = ! etatLed1; // on inverse
25        etatLed2 = ! etatLed2; // on inverse
26      }
27
28      digitalWrite(pin, etatLed1);
```

```
29        digitalWrite(pin2, etatLed2);
30    }
```

Dans cette fonction, on crée deux états etatLed1 et etatLed2, un pour chaque couleur de la led. Les deux sont initialisés à LOW. Puis avec le switch, chaque état est mis à HIGH en fonction de ce qui est demandé. On inverse si nécessaire en fonction du flag montageInverse, et on fini par appliquer l'état sur chaque led. De cette maniere Allumer2() et Allumer12() deviendront :

```
1     void LedBicouleur::Allumer2()
2     {
3       etat = HIGH2;
4       Rafraichir();
5     }
6
7     void LedBicouleur::Allumer12()
8     {
9       etat = HIGH12;
10      Rafraichir();
11    }
```

Un nouveau problème se pose. Les données **pin** et **etat** de la classe Led sont privées (**private**). LedBicouleur n'a pas le droit de les lire et de les changer. Pour que ce code puisse fonctionner, elles doivent devenir visibles pour les classes dérivées. C'est le rôle du modificateur **protected** (protégé) qui remplace **private** et ne laisse les données et les fonctions visibles que pour les classes dérivées.

```
1     class Led
2     {
3     protected:
4       byte pin;
5       byte etat;
6       bool montageInverse;
7       virtual void Rafraichir();
8
9     public:
10      Led(byte aPin, bool aMontageInverse);
11      void Allumer();
12      void Eteindre();
13    };
```

Un exemple d'utilisation de cette classe :

```
1     LedBicouleur maLed(10, 11);
2
3     void setup()
4     {
5       maLed.Setup();
6     }
7
8     void loop()
9     {
10      maLed.Allumer();
11      delay(1000);
12      maled.Eteindre();
```

```
13        maLed.Allumer2();
14        delay(1000);
15        maLed.Eteindre();
16        delay(1000);
17    }
```

VI. Pratiques de la programmation des classes

En C++, on utilse deux types de fichiers :

— Les déclarations se font dans fichiers **header** (extension .h ou .hpp ou pas d'extension),

— Les définitions se font dans fichiers d'**implémentation** (.cpp).

en général à chaque .h correspond un .cpp comme le montre la figure 9.2.

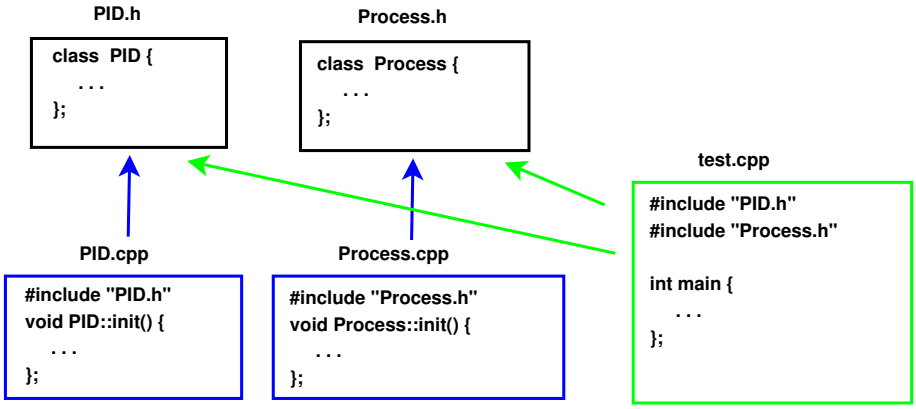


FIGURE 9.2 – Fichiers *.h* et *.cpp*

Dans le header PID.h :

```
1  class PID {
2  private:
3    float erreur;
4    float derivee;
5    float integrale;
6  public:
7    PID(float Kp, float Ki, float Kd);  // Constructeur
8    void setParams(float Kp, float Ki, float Kd); // méthode
9    ....
10 };
```

Bibliographie

[1] Bjarne Stroustrup, "**The C++ Programming Language - The Fourth Edition**", Addison-Wesley, 2013

[2] Scott Meyers, "**Effective Modern C++**", O'Reilly, 2015

[3] Jean-Michel Réveillac, "**Mini manuel de C++**", Dunod, 2010

[4] Julien Le Corre, Yannick Gerometta, "**C++ le guide complet**", MicroApplication, 2014

[5] Christine Eberhardt, "**Tout sur le C++**", Dunod, 2009

[6] `https://cpp.developpez.com/cours/cpp/`

[7] `https://www.tutorialspoint.com/cplusplus/index.htm`

[8] `www.cplusplus.com/doc/tutorial/`

[9] `https://www.learncpp.com/`

[10] `https://www.cprogramming.com/tutorial/c++-tutorial.html`