République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

UNIVERSITE MUSTAPHA STAMBOULI DE MASCARA

FACULTÉ DES SCIENCES ET TECHNOLOGIE

Polycopié de Cours

Programmation en C++

Ce cours est destiné aux étudiants de 3ème année

Licence en Automatique

Dr .BENAISSA KADDAR Leila

Algérie
2022-2023

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 permettra à l'étudiant de se familiariser avec les langages de programmation et en particulier le 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. Ce cours ne prétend ni à l'exhaustivité ni à servir de manuel complet pour une formation autodidacte. Il s'agit simplement d'un outil pédagogique qui peut être utilisé pour un premier cours présentiel : cours d'initiation à la programmation en langage C^{++}.

Table des matières

Chapitre 4. Entrées/sorties

Chapitre 5. Pointeurs et Tableaux

Chapitre 6. Les fonctions

ANNEXE

Liste des Tables

Liste des Figures

Introduction Générale

C++ est un langage de programmation populaire et puissant utilisé dans de nombreux domaines, tels que les jeux, les systèmes d'exploitation, la finance, les simulations scientifiques et plus encore. Il offre une combinaison unique de contrôle de la mémoire bas niveau et de fonctionnalités de haut niveau, ce qui le rend adapté à une large gamme de tâches de développement. Cependant, cette puissance peut également rendre le code plus difficile à écrire et à maintenir correctement, nécessitant une bonne compréhension des concepts de programmation fondamentaux et une attention particulière à la gestion de la mémoire.

En plus des points mentionnés précédemment, il convient de souligner que C++ est un langage de programmation de bas niveau qui fournit un grand contrôle sur les ressources système, ce qui en fait un choix populaire pour les applications exigeantes en termes de performances. De plus, C++ est un langage standardisé, ce qui signifie que les développeurs peuvent être assurés que leur code sera compatible avec d'autres systèmes et plateformes.

C++ est également très largement utilisé dans l'industrie, ce qui signifie qu'il existe de nombreuses ressources et bibliothèques disponibles pour les développeurs, ainsi que de nombreuses opportunités de carrière pour ceux qui maîtrisent le langage.

Dans ce cours, vous découvrirez les fondamentaux de la programmation en C++, qui s'articule autour de huit chapitres essentiels.

Le premier chapitre, présente le langage C++, son historique et les outils nécessaires au programmeur.

Le deuxième chapitre, évoque la syntaxe élémentaire du langage C++, exemple : Les Différents Types De Variables, les opérateurs arithmétiques et logiques ainsi que les opérateurs d'entrées-sorties.

Le troisième chapitre présente les différentes structures conditionnelles et boucles.

Le quatrième chapitre détaille les entrées sorties de flux de données.

Le cinquième chapitre initie la manipulation des tableaux ainsi que les pointeurs

Le sixième chapitre évoque l'utilisation des fonctions

Le septième chapitre présente la communication du programme avec d'autres fichiers texte ou autre.

Le dernier chapitre introduit la programmation orientée objet en C++.

Comprendre ces concepts de base est crucial pour la programmation en C++, et il est important de les étudier en profondeur pour maîtriser le langage.

Notre objectif est de vous donner les compétences nécessaires pour écrire du code C++ de manière efficace et pour résoudre les problèmes de programmation courants.

Alors, prêt à démarrer votre aventure dans le monde de la programmation en C++ ? Commençons !

CHAPITRE 1

PRESENTATION DU LANGAGE C++

Note

Les exercices ont été testés avec les outils BORLAND C++ BUILDER (toute version) en mode « console » et BC5.

1. Historique du C++ :

Dans les années 1970. À cette époque, **Dennis Ritchie**, programmeur aux laboratoires AT&T aux États-Unis, invente **le langage C,** conçu pour programmer le système d'exploitation UNIX. Ce langage devint très populaire à tel point qu'il est encore beaucoup utilisé aujourd'hui, dans l'écriture de Linux par exemple. Puis un peu plus tard, au début des années 1980, **Bjarne Stroustrup**, lui aussi développeur aux laboratoires AT&T, décida de prendre le langage C comme base et de lui ajouter des fonctionnalités issues d'un autre langage appelé Simula. Ce langage devint alors le **C with classes**.

Finalement, en 1983, son créateur, estimant que le nom de son langage était trop réducteur aux vues de tous les ajouts faits par rapport au C, décida de le renommer **C++**. Mais l'histoire ne s'arrête pas là. Au contraire, le C++ continue d'évoluer à tel point qu'on décide au début des années 1990 de **le normaliser**, c'est-à-dire d'en établir les règles officielles. Ce travail de longue haleine s'acheva en 1998 ; cette version est ainsi souvent nommée C++98. Ensuite, en 2003, des corrections ont été apportées et l'on obtint C++03. Puis de nouveau un chantier titanesque est mis en place pour améliorer encore plus le C++, ce qui aboutit 8 ans plus tard, en 2011, à la sortie de C++11, jugée par beaucoup de développeurs comme étant la renaissance du C++. Ensuite, de nouvelles corrections et quelques ajustements ont été apportés pour donner C++14. Enfin, à l'heure actuelle, la norme C++17, nouvelle version majeure et apportant tout un lot de choses intéressantes, est sortie en fin d'année et C++20 est déjà en chantier.

Il faut savoir que beaucoup de programmeurs utilisent le terme «C++ historique» pour désigner les normes C++98 et C++03 et le terme «C++ moderne» pour parler de C++11 et au-delà. Après ce bref historique une question vient à l'esprit, pourquoi apprendre le C++. La réponse peut se résumer dans les points suivants :

- Sa **popularité** : le C++ est un langage qui est utilisé dans de nombreux projets important (citons **Libre Office, 7-zip** ou encore **KDE**). Il est au programme de beaucoup de formations informatiques. Il possède une communauté très importante, beaucoup de documentation et d'aide, surtout sur l'internet anglophone.
- Sa **rapidité** : C++ offre un grand contrôle sur la rapidité des programmes. C'est cette caractéristique qui fait de lui un des langages de choix pour les programmes scientifiques, par exemple.
- Sa **facilité d'apprentissage** : depuis sa version de 2011, C++ est beaucoup plus facile à apprendre que par le passé. Et ça tombe bien, c'est sur cette version et les suivantes que va se baser ce cours.

- Son **ancienneté** : C++ est un langage ancien d'un point de vue informatique (30 ans, c'est énorme), ce qui donne une certaine garantie de maturité, de stabilité et de pérennité (il ne disparaîtra pas dans quelques années).
- Son **évolution** : C++11 est un véritable renouveau de C++, qui le rend plus facile à utiliser et plus puissant dans les fonctionnalités qu'il offre aux développeurs. C++14 et C++17 améliorent encore la chose.
- Il est **multi-paradigme** : il n'impose pas une façon unique de concevoir et découper ses programmes mais laisse le développeur libre de ses choix, contrairement à d'autres langages comme **Java** ou **Haskell.**

Bien entendu, tout n'est pas parfait et C++ a aussi ses défauts.

- Son **héritage du C** : C++ est un descendant du langage C, inventé dans les années 1970. Certains choix de conception, adaptés pour l'époque, sont plus problématiques aujourd'hui, et C++ les traine avec lui.
- Sa **complexité** : il ne faut pas se le cacher, avoir une certaine maîtrise du C++ est très long et demandera des années d'expérience, notamment parce que certaines des fonctionnalités les plus puissantes du C++ requièrent de bien connaître les bases.

2. Environnement De Développement En C++ :

Il faut installer certains logiciels spécifiques afin de développer des applications en C++.

2.1. Les outils nécessaires au programmeur :

Il s'agit du compilateur, ce fameux programme qui permet de traduire votre langage C++ en langage binaire.

Il existe plusieurs compilateurs pour le langage C++:

- Visual C++ (Windows seulement).
- Visual C++ Express.
- Xcode (Mac OS seulement).
- BORLAND C++

- Turbo C++
- Code::blocks (Windows, Mac OS, Linux).

2.2. Création et l'Exécution du programme d'un programme en C++ :

2.2.1. Edition du programme source, à l'aide d'un éditeur (traitement de textes). Le nom du fichier contient l'extension .CPP, EX_1.CPP (menu « edit »).

2.2.2. Compilation du programme source, c'est à dire création des codes machine destinés au microprocesseur utilisé. Le compilateur indique les erreurs de syntaxe mais ignore les fonctions-bibliothèque appelées par le programme.

Le compilateur génère un fichier binaire, non éditable en mode « texte », appelé fichier objet : EX_1.OBJ (commande « compile »).

2.2.3. Editions de liens : Le code machine des fonctions-bibliothèque est chargé, création d'un fichier binaire, non éditable en mode texte, appelé fichier exécutable : EX_1.EXE (commande « build all »).

2.2.4. Exécution du programme (commande « Run » ou « flèche jaune »).

Les compilateurs permettent en général de construire des programmes composés de plusieurs fichiers sources, d'ajouter à un programme des unités déjà compilées. On dit alors que l'on travaille par gestion de projet.

à savoir

Le langage C++ possède assez peu d'instructions, il fait par contre appel à des bibliothèques, fournies en plus ou moins grand nombre avec le compilateur.

Exemples : **math.h :** bibliothèque de fonctions mathématiques

iostream.h : bibliothèque d'entrées/sorties standard

complex.h : bibliothèque contenant la classe des nombres complexes.

On ne saurait développer un programme en C++ sans se munir de la documentation concernant ces bibliothèques.

Un premier exemple

Nous allons maintenant étudier ce premier exemple (EX-1) :

```
1  #include <iostream.h> // sorties standards
2  #include <conio.h> // les commentaires s'écrivent derrière 2 barres
3  void main()
4  {
5  cout<<"BONJOUR";//affichage d'un message sur l'écran
6  cout<<" Belle journée!!";//affichage d'un autre message sur l'écran
7  cout<<"Pour continuer frapper une touche...";
8  // Attente d'une saisie clavier pour voir l'écran d'exécution
9  getch();
10 }
```

La directive #include :

On place en général au début du programme un certain nombre d'instructions commençant par #include. Cette instruction permet d'inclure dans un programme la définition de certains objets, types ou fonctions. Le nom du fichier peut être soit à l'intérieur des chevrons < et >, soit entre guillemets :

- #include <nom_fichier> Inclut le fichier nom_fichier en le cherchant d'abord dans les chemins configurés, puis dans le même répertoire que le fichier source,
- #include "nom_fichier" Inclut le fichier nom_fichier en le cherchant d'abord dans le même répertoire que le fichier source, puis dans les chemins configurés.

Le fichier iostream :

Le fichier iostream contient un certain nombre de définitions d'objets intervenant dans les entrées/sorties du programme, c'est-à-dire dans l'affichage à l'écran ou dans des fichiers. La définition de cout se trouve dans ce fichier ; pour utiliser cout dans notre programme, il faut inclure au début du programme la ligne suivante :

- # include <iostream> Ce fichier est fourni par l'éditeur du compilateur : il s'agit d'un fichier C++ standard.

La fonction main () :

Notre programme contient une fonction appelée main : c'est à cet endroit que va commencer l'exécution du programme : exécuter un programme en C++, c'est exécuter la fonction main de ce programme. Tout programme en C++ doit donc comporter une fonction main.

Cout :

Il s'agit du flux de sortie du programme (Console Output : sortie console). Ce flux de sortie est envoyé par défaut vers l'écran. Il va nous servir à afficher des messages à l'écran en utilisant l'opérateur <<. Cet opérateur à la forme d'une flèche semblant indiquer le sens de transfert des données (écriture vers la console).

Exemple : `cout<<"BONJOUR";`

Cette instruction affiche **BONJOUR** à l'écran.

La fonction getch() :

La fonction getch, appartenant à la bibliothèque **conio.h** permet la saisie clavier d'un caractère alphanumérique, sans écho écran. La saisie s'arrête dès que le caractère a été frappé.

Note

Le langage C++ distingue les minuscules, des majuscules. Les mots réservés du langage C++ doivent être écrits en minuscules.

On a introduit dans ce programme la notion d'interface homme/machine (IHM).

- L'utilisateur visualise une information sur l'écran,
- L'utilisateur, par une action sur le clavier, fournit une information au programme.

Les instructions sont exécutées séquentiellement, c'est à dire les unes après les autres. L'ordre dans lequel elles sont écrites a donc une grande importance.

CHAPITRE 2

SYNTAXE ELEMENTAIRE EN LANGAGE C++

Avant d'approfondir la programmation C++ nous allons illustrer le principe de la syntaxe élémentaire d'écriture d'un programme C++.

1. Les commentaires

Ils peuvent être faits à n'importe quel endroit du programme. Ce sont des textes explicatifs destinés aux lecteurs du programme. Ils ne sont pas pris en compte par le compilateur. On distingue deux types de commentaires :

- **Les commentaires libres (par bloc) :**

Ils commencent par /* et se termine par */. Ils peuvent s'étendre sur plusieurs lignes. Un exemple est donné ci-dessous :

```
1 /*+------------------------------------------------------------+
2  |                Mon premier programme en C++                |
3 +------------------------------------------------------------+ */
```

- **Les commentaires de ligne (de fin de ligne) :**

Ils sont introduit par les deux caractères //. Ils sont généralement placés en fin de ligne après une instruction. Tout ce qui se situe entre // et la fin de la ligne est un commentaire.

```
1 cout << "Hello \n" ; // bonjour
```

2. Les Différents Types De Variables :

2.1. Les entiers :

Le langage C++ distingue plusieurs types d'entiers :

TYPE	DESCRIPTION	TAILLE MEMOIRE
int	entier standard signé	4 octets : - 231 ≤ n ≤ 231-1
unsigned int	entier positif	4 octets: 0 ≤ n ≤ 232
short	entier court signé	2 octets : - 215 ≤ n ≤ 215-1
unsigned short	entier court non signé	2 octets: 0 ≤ n ≤ 216
char	caractère signé	1 octet : - 27 ≤ n ≤ 27-1
unsigned char	caractère non signé	1 octet : 0 ≤n ≤ 28

Numération :

- En décimal les nombres s'écrivent tels que,

- En hexadécimal ils sont précédés de 0x.

Exemple : 127 en décimal s'écrit 0x7f en hexadécimal.

Remarque

En langage C++, le type char possède une fonction de changement de type vers un entier :

- Un caractère peut voir son type automatiquement transformé vers un entier de 8 bits.
- Il est interprété comme un caractère alphanumérique du clavier.

Exemples :

- Les caractères alphanumériques s'écrivent entre ' '
- Le caractère 'b' a pour valeur 98.
- Le caractère 22 a pour valeur 22.
- Le caractère 127 a pour valeur 127.
- Le caractère 257 a pour valeur 1 (ce nombre s'écrit sur 9 bits, le bit de poids fort est perdu).

Quelques constantes caractères :

CARACTERE	VALEUR (code ASCII)	NOM ASCII
'\n' interligne	0x0a	LF
'\t' tabulation horizontale	0x09	HT
'\v' tabulation verticale	0x0b	VT
'\r' retour chariot	0x0d	CR
'\f' saut de page	0x0c	FF
'\\' backslash	0x5c	\
'\'' cote	0x2c	'
'\"' guillemets	0x22	"

2.2. Les réels :

Un réel est composé :

- d'un signe,
- d'une mantisse,
- d'un exposant,

Un nombre de bits est réservé en mémoire pour chaque élément.

Le langage C++ distingue 2 types de réels :

TYPE	DESCRIPTION	TAILLE MEMOIRE
float	réel standard	4 octets
double	réel double précision	8 octets

Les Initialisations :

Le langage C++ permet l'initialisation des variables dès leurs déclarations :

`char c ;` est équivalent à `char c = 'A';`

`c = 'A';`

`int i;` est équivalent à `int i = 50;`

`i = 50;`

Cette règle s'applique à tous les nombres, char, int, float ... Pour améliorer la lisibilité des programmes et leur efficacité, il est conseillé de l'utiliser.

2.3. Le type booléen :

Il contient deux valeurs true et false.

L'exemple suivant récapitule la déclaration de variables des différents types cités.

Exemple EX-2

```
1  #include <iostream.h>
2  #include <conio.h>
3  void main()
4  {
5  int i = 2;
6  float x = 4.3;
7  float y = 54e -01;
8  double z = 5.4;
9  unsigned char c = 'A';
10 getch();
11 }
```

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.
- bool : valeur booléenne.

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.h> // sorties standards
2  #include <conio.h> // les commentaires s'écrivent derrière 2 barres
3  void main()
4  {
5  cout<<"TAILLE D'UN CARACTERE : "<<sizeof(char)<< "\n";
6  cout<<"TAILLE D'UN ENTIER : " <<sizeof(int)<< "\n";
7  cout<<"TAILLE D'UN REEL : " <<sizeof(float)<< "\n";
8  cout<<"TAILLE D'UN DOUBLE : " <<sizeof(double)<< "\n";
9  cout<<"TAILLE D'UN DOUBLE : " <<sizeof(bool)<< "\n";
10 cout<<"Pour continuer frapper une touche...";
11 // Attente d'une saisie clavier pour voir l'écran d'exécution
12 getch();
13 }
```

Ce programme donne en sortie :

sizeof (char) = 1

sizeof(int) = 4

sizeof(float) = 4

sizeof(double) = 8

sizeof(bool) = 1

3. Les déclarations de constantes :

Le langage C++ autorise 2 méthodes pour définir des constantes.

1ere méthode : déclaration d'une variable, dont la valeur sera constante pour toute la portée de la fonction main. (Voir exemple suivant)

```
1  void main()
2  {
3  const float PI = 3.14159;
4  float perimetre, rayon = 8.7;
5  perimetre = 2*rayon*PI;
6  //…
7  }
```

Dans ce cas, le compilateur réserve de la place en mémoire (ici 4 octets), pour la variable pi, on ne peut changer la valeur. On peut associer un modificateur« const » à tous les types.

2eme méthode : définition d'un symbole à l'aide de la directive de compilation **#define.**

```
#define PI = 3.14159;
void main()
{
float perimetre, rayon = 8.7;
perimetre = 2*rayon*PI;
//…
}
```

Le compilateur ne réserve pas de place en mémoire, on définit ainsi une équivalence « lexicale ».

Les constantes déclarées par **#define** s'écrivent traditionnellement en majuscules, mais ce n'est pas une obligation.

4. Les opérateurs :

4.1. Les 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.

- **Les Opérateurs arithmétiques sur les réels :** + - * / avec la hiérarchie habituelle.
- **Les Opérateurs arithmétiques sur les entiers :** + - * / (quotient de la division) % (reste de la division) avec la hiérarchie habituelle.

Exemple particulier : char c, d ; c = 'G' ; d = c+'a'-'A';

Les caractères sont des entiers sur 8 bits, on peut donc effectuer des opérations. Sur cet exemple, on transforme la lettre majuscule G en la lettre minuscule g.

- **Les Opérateurs d'incrémentation et de décrémentation :**

Opérateur	Exemple d'utilisation	Résultat
Incrémentation	i++	i=i+1
Décrémentation	i--	i=i-1

Les opérateurs d'incrémentation et de décrémentation

- Dans le cas de l'opérateur préfixé, le ++ ou -- est placé avant la variable. Dans ce cas, la variable est incrémentée (ou décrémentée) avant l'utilisation de la valeur.
- Dans le cas de l'opérateur post fixé, le ++ ou -- est placé après la variable. Dans ce cas, la variable est incrémentée (ou décrémentée) après l'utilisation de la valeur.

L'exemple suivant montre la différence entre les deux cas.

```
1  int x=1 ;
2  int y=0 ; y=x++ ; // Dans ce cas y=1 et x=2
3   x=1 ; // On réinitialise x
4   y=++ x ; // Dans ce cas x=2 et y=2
```

4.2. Les opérateurs logiques :

- **Les opérateurs logiques sur les entiers :**

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

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

Table 1. Les Operateurs Logiques

- **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. Les opérateurs bit-à-bit

4.3. 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 décalages 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. Le premier opérande désigne la donnée sur laquelle on va faire le décalage, la seconde désigne le nombre de bits duquel elle va être décalée.

«	Décalage à gauche	6 « 1 (110 « 1)	12 (1100)
»	Décalage à droite	6 » 1 (0110 » 1)	3 (0011)

Table 3. Les opérateurs de décalage de bit

Exemples : p = n « 3 ; // p est égale à n décalé de 3 bits à gauche

p = n » 3 ; // p est égale à n décalé de 3 bits à droite

4.4. 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 5. Les opérateurs de comparaison

4.5. Operateurs Combinés :

Le langage C++ autorise des écritures simplifiées lorsqu'une même variable est utilisée de chaque côté du signe = d'une affectation. Ces écritures sont à éviter lorsque l'on débute l'étude du langage C++ car elles nuisent à la lisibilité du programme.

a = a+b; est équivalent à **a+= b**;

a = a-b; est équivalent à **a-= b**;

a = a & b; est équivalent à **a&= b**;

4.6. 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 (conversions 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.

4.7. Opérateurs d'entrées-sorties :

On peut utiliser en C++ les fonctions d'entrées/sorties classiques du C (printf, scanf, puts, gets, putc, getc ...), à condition de déclarer le fichier d'en-tête stdio.h.

Il existe de nouvelles possibilités en C++, à condition de déclarer le fichier d'en-tête iostream.h.

- **Sortie sur écran : l'opérateur cout :**

Exemples :
```
cout << "BONJOUR" ; // équivalent à puts("BONJOUR");
int n = 25 ;
cout << "Valeur: "; // équivalent à puts("Valeur");
cout << n; // équivalent à printf("%d",n);
```

- **Saisie clavier : l'opérateur cin:**

L'opérateur `cin` permet de saisir des nombres entiers ou réels, des caractères, des chaînes de caractères.

Exemples :
```
int n;
cout<<"Saisir un entier : ";
cin >> n ; // équivalent à scanf("%d",&n);
int a, b ;
cin >> a >> b ; // saisie de 2 entiers séparés par un Retour Charriot
```

CHAPITRE 3

STRUCTURES CONDITIONNELLES ET BOUCLES

1. Introduction :

Un programme écrit en C++ s'exécute séquentiellement, c'est à dire instruction après instruction.

Ce chapitre explique comment, dans un programme, on pourra :

- ne pas exécuter une partie des instructions, c'est à dire faire un saut dans le programme,
- revenir en arrière, pour exécuter plusieurs fois de suite la même partie d'un programme.

2. Les opérateurs logiques entre les expressions :

- condition d'égalité : if (a==b) " si a est égal à b "
- condition de non égalité : if (a!=b) " si a est différent de b "
- conditions de relation d'ordre : if (a<b) if (a<=b) if (a>b) if (a>=b)

Plusieurs conditions devant être vraies simultanément, ET LOGIQUE :

- if ((expression1) && (expression2)) " si l'expression1 ET l'expression2 sont vraies

Une condition devant être vraie parmi plusieurs, OU LOGIQUE :

- if ((expression1) || (expression2)) " si l'expression1 OU l'expression2 est vraie "

condition fausse if (!(expression1)) " si l'expression1 est fausse "

Toutes les combinaisons sont possibles entre ces tests.

3. Différentes formes de l'instruction if :

3.1. Instruction if :

La forme générale d'une déclaration if est :

```
if (expression)
{
      Déclaration-intérieur ;
}
      Déclaration -extérieur ;
```

Si l'expression est vraie, alors "Déclaration-intérieur" elle sera exécutée, sinon

'Déclaration intérieur' est ignoré et seulement " Déclaration- extérieur" est exécuté.

Exemple :

```
1
2  #include <iostream.h>
3  #include <conio.h>
4  void main()
5  {
6  int x,y;
7  x =15;
8  y =13;
9  if (x > y )
10   {
11   cout << "x est supérieur à y"<<"\n";
12   }
13 getch();
   }
```

3.2. 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.h>
#include <conio.h>
void main()
{
int x,y;
x =14;
y =19;
if (x > y)
  {
  cout << "x est supérieur à y"<<"\n";
  } else
{
  cout << "y est supérieur à x"<<"\n";
  }
getch () ;
}
```

3.3. 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 <iostream.h>
#include <conio.h>
void main()
{
int a,b,c;
cout << " entrez 3 nombres "<< "\n";
 cin >> a >> b >> c;
if(a>b)
{
if( a > c)
 {
cout << "a est le plus grand "<< "\n";
}
else
{
cout << "c est le plus grand " <<"\n";
}
}
else
{
 if( b> c)
 {
cout << "b est le plus grand " <<"\n";
 }
  else
 {
 cout << "c'est le plus grand " <<"\n";
}
}
getch () ;
 }
```

3.4. 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.

```
1  #include <iostream.h>
2  #include <conio.h>
3  void main()
4  {
5  int a;
6   cout << " entrez un nombre " <<"\n";
7  cin >> a;
8   if( a %5==0 && a %8==0)
9   {
10  cout <<" divisible par les 5 et 8" <<"\n";
11 }
12 else if( a %8==0 )
13  {
14  cout <<" divisible par 8" <<"\n";
15 }
16 else if(a %5==0)
17 {
18 cout <<" divisible par 5" <<"\n";
19 }
20  else
21  {
22 cout <<"non divisible " <<"\n";
23 }
24  getch ();
25 }
```

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.

4. Les Boucles :

Les boucles vous permettent de répéter les mêmes instructions plusieurs fois dans votre programme. Le principe est le suivant :

- L'ordinateur lit les instructions de haut en bas (comme d'habitude)
- Puis, une fois arrivé à la fin de la boucle, il repart à la première instruction
- Il recommence alors à lire les instructions de haut en bas...
- ... Et il repart au début de la boucle.

Les boucles sont répétées tant qu'une condition est vraie. Par exemple on peut faire une boucle qui dit : "Tant que l'utilisateur donne un nombre d'enfants inférieur à 0, redemander le nombre d'enfants"...

Il existe 3 types de boucles à connaître :

- **While**
- **do ... while**
- **for**

4.1. La boucle while :

Il s'agit de l'instruction : tant que (condition vraie)

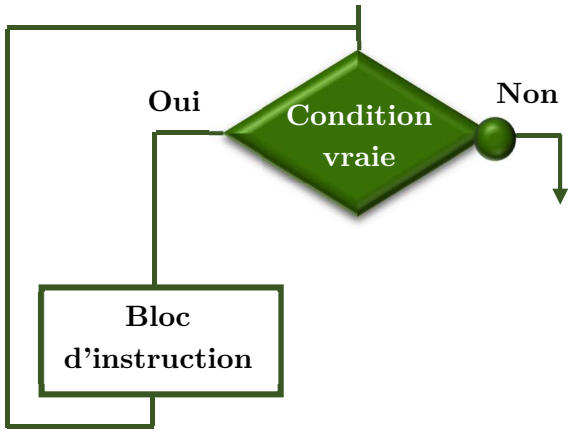
Exécuter

{

BLOC D'INSTRUCTIONS

}

Organigramme :



Syntaxe en C++:

```
while (condition)
{
……..; // bloc d'instructions
……..;
……..;
}
suite du programme ...
```

Le test se fait d'abord, le bloc d'instructions n'est pas forcément exécuté.

Remarque

- Les {} ne sont pas nécessaires lorsque le bloc ne comporte qu'une seule instruction.
- On peut rencontrer la construction suivante : while (expression) ; terminée par un ; et sans la présence du bloc d'instructions. Cette construction signifie : "tant que l'expression est vraie attendre". Ceci ne doit être exploité que par rapport à des événements externes au programme (attente de la frappe du clavier par exemple).

Exemple :

On redemande le nombre d'enfants à l'utilisateur tant que celui-ci est inférieur à 0. Ce genre de boucle permet de s'assurer que l'utilisateur rentre un nombre correct.

```
1  #include <iostream.h>
2  #include <conio.h>
3  void main()
4  {
5  int nbEnfants(-1); // Nombre négatif pour pouvoir rentrer dans la boucle
6  while (nbEnfants < 0)
7  {
8  cout << "Combien d'enfants avez-vous ?" <<"\n";
9  cin >> nbEnfants;
10 }
11 cout << "Merci d'avoir indique un nombre d'enfants correct. Vous en avez" <<
12 nbEnfants <<"\n";
13 getch ();
14 }
```

Tant que vous mettrez un nombre négatif, la boucle recommencera. En effet, elle teste while (nbEnfants < 0) c'est-à-dire "Tant que le nombre d'enfants est inférieur à 0". Dès que le nombre devient supérieur ou égal à 0, la boucle s'arrête et le programme continue après l'accolade fermante.

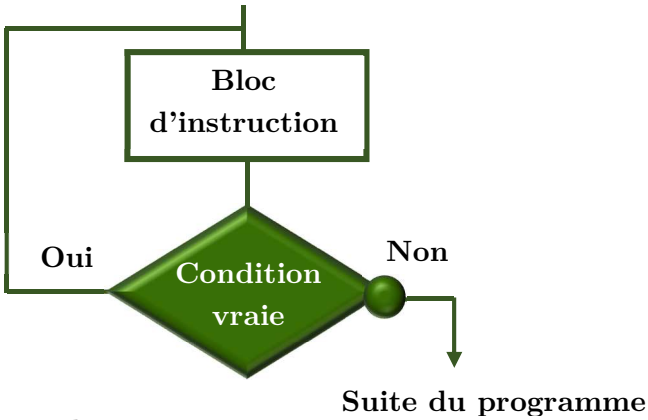
4.2. La boucle do while :

Cette boucle est très similaire à la précédente... si ce n'est que la condition n'est analysée qu'à la fin. Cela signifie que le contenu de la boucle sera toujours lu au moins une première fois.

Il s'agit de l'instruction :

exécuter
{
BLOC D'INSTRUCTIONS
}
while (condition vraie)

Organigramme :



Syntaxe en C++:

```
do
{
.....; // bloc d'instructions
.....;
.....;
}
while (condition);
suite du programme ...
```

Le test se faisant après, le bloc est exécuté au moins une fois.

Exemple :

Reprenons le code de l'exemple précédent et utilisons cette fois un do ... while :

```
1  #include <iostream.h>
2  #include <conio.h>
3  void main()
4  {
5  int nbEnfants (0);
6  do
7  {
8  cout << "Combien d'enfants avez-vous ?" <<"\n";
9  cin >> nbEnfants;
10 } while (nbEnfants < 0);
11 cout << "Merci d'avoir indique un nombre d'enfants correct. Vous en avez"
12 << nbEnfants <<"\n";
13 getch ();
14 }
```

Le principe est le même, le programme a le même comportement. Le gros intérêt du do ... while est qu'on s'assure que la boucle sera lue au moins une fois.

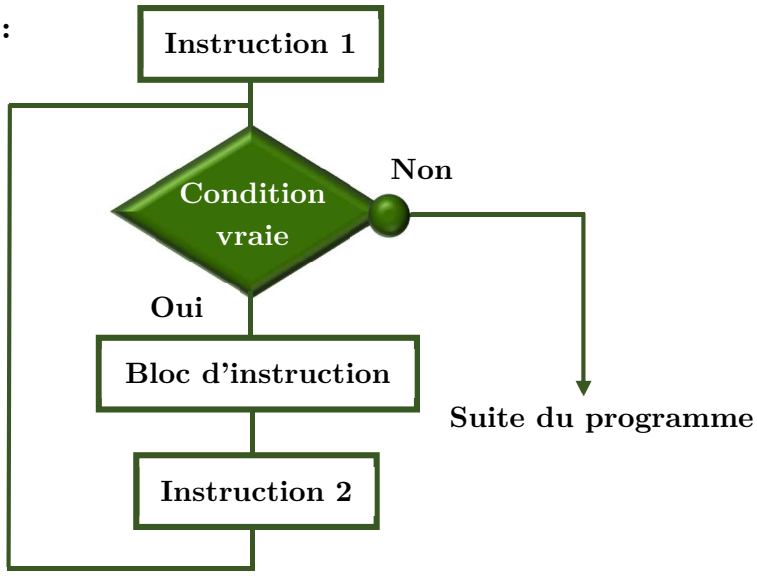
D'ailleurs, du coup, il n'est pas nécessaire d'initialiser nbEnfants à -1 (c'est le principal avantage que procure cette solution). En effet, ici le nombre d'enfants est initialisé à 0 (comme on a l'habitude de faire), et comme la condition n'est testée qu'après le premier passage de la boucle, les instructions sont bien lues au moins une fois.

4.3. 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 (instruction1 ; condition ; instruction2)
{
BLOC D'INSTRUCTIONS
}

Organigramme :



Syntaxe en C++ :

```
for(instruction1; condition; instruction2)
{
.....; // bloc d'instructions
.....;
.....;
}
suite du programme ...
```

Remarque

Il s'agit d'une version enrichie du « while ».

Les {} ne sont pas nécessaires lorsque le bloc ne comporte qu'une seule instruction.

Les 3 instructions du for ne portent pas forcément sur la même variable.

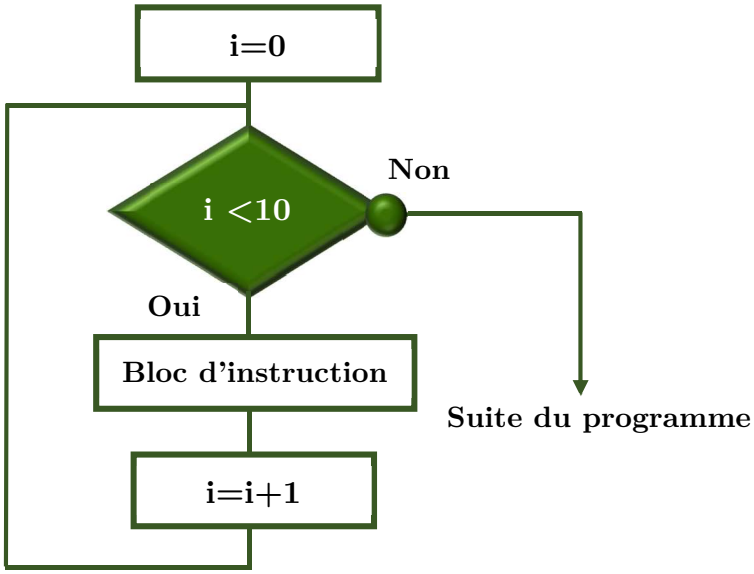
Une instruction peut être omise, mais pas les ;

Exemples :

Voici un exemple concret qui affiche des nombres de 0 à 10

```
1  #include <iostream.h>
2  #include <conio.h>
3  void main()
4  {
5  int compteur(0);
6  for (compteur = 0 ; compteur < 10 ; compteur++)
7  {
8  cout << compteur <<"\n";
9  }
10 getch ();
11 }
```

Correspond à l'organigramme suivant :



Exemple :

Voici un exemple concret qui affiche des nombres de 0 à 9 :

La boucle **for(;;)**

```
{
.....; // bloc d'instructions
.....;
.....;
}
```

est une boucle infinie (répétition infinie du bloc d'instructions).

Utilisation de variables différentes :

```
resultat = 0;
for(int i = 0 ; resultat<30 ; i++)
{
.....; // bloc d'instructions
.....;
.....;
resultat = resultat + 2*i;
}
```

5. Contrôler plus finement l'exécution de la boucle :

Les boucles sont très utiles, mais le code à l'intérieur des accolades est soit entièrement sauté, soit entièrement exécuté. Sachez que C++ permet de contrôler un peu plus précisément l'exécution avec deux mots-clefs : `break` et `continue`.

Précision

Simple précision : **break** et **continue** s'utilisent avec les boucles uniquement. Ils ne s'utilisent pas avec les conditions : pas de **break** au sein d'un **if**, par exemple.

5.1. break — Je m'arrête là

Le premier, c'est **break**, qui en anglais signifie littéralement «casser», «interrompre». Celui-ci nous autorise à **mettre fin à l'exécution de la boucle**, peu importe où nous en étions. Dans le code ci-dessous, vous ne verrez jamais afficher les nombres au-dessus de 3 parce que la boucle est terminée par **break** dès que notre variable vaut 3.

```
#include <iostream.h>
#include <conio.h>
void main()
{
for (int i { 0 }; i < 10; ++i)
{
// Si i vaut 3, j'arrête la boucle.
if (i == 3)
{
break;
}
cout << i <<"\n";
}
getch ();
}
```

5.2. Continue — Saute ton tour !

L'autre mot-clef, continue, permet de sauter l'itération courante. Toutes les instructions qui se trouvent après ce mot-clef sont donc ignorées et la boucle continue au tour suivant.

Imaginons que nous voulions un programme permettant d'afficher les nombres impairs compris entre 0 et un nombre choisi par l'utilisateur. Essayez donc de le faire, avec les connaissances que vous avez.

5.3. 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 ( expression )
{
case val1 : // instruction1 ou bloc d'instructions1
break ;
case val2 : // instruction2 ou bloc d'instructions2
break ;
 ...
default : // instruction ou bloc d'instructions par d´efaut
 }
```

L'expression est évaluée à l'entrée de la boucle switch. Elle doit donner un résultat de type entier. val1, val2 sont des constantes de type entier. L'exécution de l'instruction se fait à partir du case dont la valeur correspond à celle de l'expression évaluée. L'exécution peut continuer pour les autres case jusqu'à rencontrer un break. Si la valeur de l'expression ne correspond à aucune des valeurs des case, c'est les instructions de default qui sont exécutée. Un exemple d'utilisation de l'instruction switch-case est donné par le programme suivant :

```
#include <iostream.h>
#include <conio.h>
void main()
{
int i;
cout <<" Entrez 1 à 4 pour afficher une lettre de l'alphabet "<<"\n";
cin >>i;
switch (i)
 {
case 1 :
cout <<"A"<<"\n";
break ;
case 2 :
 cout <<"B"<<"\n";
break ;
case 3 :
cout <<"C"<<"\n";
break ;
case 4 :
cout <<"D"<<"\n";
break ;
default :
cout <<" Erreur "<<"\n";
}
getch ();
}
```

CHAPITRE 4

ENTREES/SORTIES

1. Introduction :

En C++, les entrées/sorties (E/S) sont des opérations cruciales pour les applications qui interagissent avec l'utilisateur ou les données en dehors du programme. Les entrées/sorties permettent d'entrer des données dans le programme à partir de sources externes telles que le clavier ou un fichier, et de sortir les résultats du programme vers des destinations telles que l'écran ou un fichier.

Le C++ fournit des moyens de gérer les entrées/sorties à l'aide de la bibliothèque standard iostream, qui est incluse dans la plupart des compilateurs C++. Les entrées/sorties peuvent être effectuées à l'aide d'opérateurs et de fonctions standard tels que cin (entrée standard) et cout (sortie standard). Les entrées/sorties peuvent également être effectuées à l'aide de fichiers en utilisant des classes telles que fstream (fichiers en entrée/sortie).

L'utilisation des entrées/sorties en C++ peut être très utile pour les applications interactives telles que les jeux ou les applications de base de données, car elles permettent de recevoir des données d'utilisateur et de les afficher de manière conviviale. Les entrées/sorties peuvent également être utilisées pour enregistrer et

charger des données dans des fichiers, ce qui peut être très utile pour les applications qui doivent mémoriser les données entre les sessions d'exécution.

2. Entrée/Sortie de base en C++ :

C++ est livré avec des bibliothèques qui nous offrent de nombreuses façons d'effectuer des entrées et des sorties. En C++, l'entrée et la sortie sont effectuées sous la forme d'une séquence d'octets ou plus communément appelées flux.

- Flux d'entrée : si le sens du flux d'octets va du périphérique (par exemple, le clavier) à la mémoire principale, ce processus est appelé entrée.
- Flux de sortie : si le sens de circulation des octets est opposé, c'est-à-dire de la mémoire principale à l'appareil (écran d'affichage), ce processus est appelé sortie.

2.1. Classes de gestion des flux :

Les entrées et sorties sont gérées par deux classes définies dans le fichier d'en-tête <iostream> :

- **ostream** (Output stream) permet d'écrire des données vers la console, un fichier, ... Cette classe surdéfinit l'opérateur <<.
- **istream** (Input stream) permet de lire des données à partir de la console, d'un fichier, ... Cette classe surdéfinit l'opérateur >>.

2.2. Les différents types de flux :

2.2.1. Le flux de sortie standard (cout) :

L'objet cout prédéfini est une instance de la classe ostream. L'objet cout est dit « connecté » au périphérique de sortie standard, qui est habituellement l'écran d'affichage. « cout » est utilisé en conjonction avec l'opérateur d'insertion de flux, qui est écrit comme << qui sont deux signes d'inférieur.

2.2.2. Le flux d'entrée standard (cin) :

L'objet prédéfini cin est une instance de la classe istream. L'objet cin est attaché au périphérique d'entrée standard, qui est habituellement le clavier. Le cin est utilisé en

conjonction avec l'opérateur d'extraction de flux, qui est écrit comme >> qui sont deux signes de supérieur.

2.2.3. Le flux d'erreur standard (cerr) :

L'objet prédéfini cerr est une instance de la classe ostream. L'objet cerr est attaché au dispositif d'erreur standard, qui est également un écran d'affichage mais l'objet cerr n'est pas tamponné et chaque insertion de flux à cerr fait que sa sortie apparaisse immédiatement.

2.2.4. Le flux log standard (clog) :

L'objet prédéfini clog est une instance de la classe ostream. On dit que l'objet clog est attaché au dispositif d'erreur standard, qui est également un écran d'affichage, mais que l'objet clog est tamponné. Cela signifie que chaque insertion à clog pourrait faire en sorte que sa sortie soit maintenue dans un tampon jusqu'à ce que le tampon soit rempli ou jusqu'à ce que le tampon soit purgé.

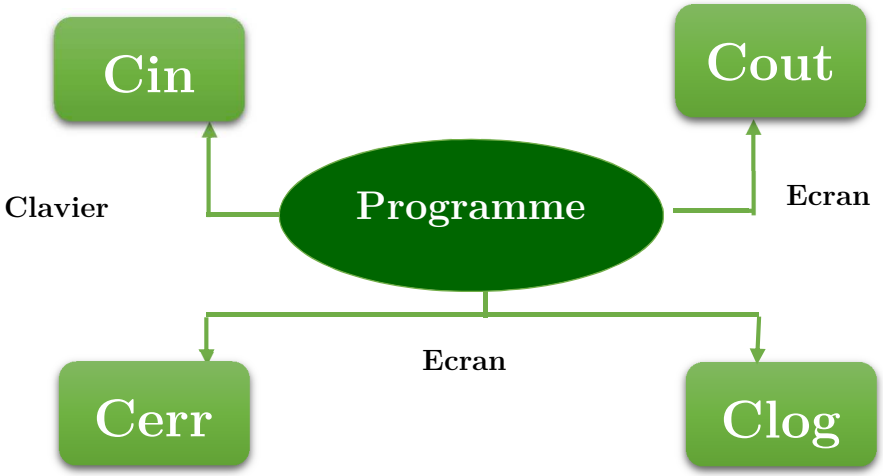


Figure 1. Les différents types de flux

3. Les chaines de caractères :

En langage C++, les chaînes de caractères sont des tableaux de caractères. Leur manipulation est donc analogue à celle d'un tableau à une dimension :

Déclaration : char nom[dim]; ou bien char *nom=new char[dim];

Exemple : char texte [20] ; ou bien char *texte=new char [20];

Il faut toujours prévoir une place de plus pour une chaîne de caractères. En effet, en C ou en

C++, une chaîne de caractère se termine toujours par le caractère NUL ('\0'). Ce caractère permet de détecter la fin de la chaîne lorsque l'on écrit des programmes de traitement.

Affichage à l'écran :

On utilise l'opérateur cout :

```
char texte[10] = « BONJOUR »;
cout<<"VOICI LE TEXTE:"<<texte;
```

Saisie :

On utilise l'opérateur cin :

```
char texte[10];
cout<<"ENTRER UN TEXTE: ";
cin>> texte;
```

Remarque

cin ne permet pas la saisie d'une chaîne comportant des espaces : les caractères saisis à partir de l'espace ne sont pas pris en compte (l'espace est un délimiteur au même titre que LF).

A l'issue de la saisie d'une chaîne de caractères, le compilateur ajoute '\0' en mémoire après le dernier caractère.

Générales (**string.h**):

Nom : **strcat**

Prototype : **void *strcat(char *chaine1, char *chaine2);**

Fonctionnement : concatène les 2 chaînes, résultat dans chaine1, renvoie l'adresse de chaine1.

Exemple d'utilisation :

```
1  char texte1[30] = "BONJOUR "; // remarquer l'espace après le R
2  char texte2[20]= "LES AMIS";
3  strcat(texte1,texte2);   // texte2 est inchangée, texte1 vaut maintenant
                               "BONJOUR LES AMIS"
```

Nom : strlen

Prototype : **int strlen(char *chaine);**

Fonctionnement : envoie la longueur de la chaîne ('\0' non comptabilisé).

Exemple d'utilisation :

```
1  char texte1[30] = "BONJOUR "; // remarquer l'espace après le R
2  int L = strlen(texte1);
3  cout<< "longueur de la chaîne : "<<L ; // L vaut 7
```

Nom: strrev

Prototype : **void *strrev(char *chaine);**

Fonctionnement : inverse la chaîne et, renvoie l'adresse de la chaîne inversée.

Exemple d'utilisation :

```
1  char texte1[10] = "BONJOUR";
2  strrev(texte1); // texte1 vaut maintenant "RUOJNOB"
```

Comparaison (string.h):

Nom : strcmp

Prototype : **int strcmp(char *chaine1, char *chaine2);**

Fonctionnement : renvoie un nombre :

- positif si chaine1 est supérieure à chaine2 (au sens de l'ordre alphabétique)

- négatif si chaîne1 est inférieure à chaine2

- nul si les chaînes sont identiques.

Cette fonction est utilisée pour classer des chaînes de caractères par ordre alphabétique.

Exemple d'utilisation :

```
1  char texte1[30] = "BONJOUR ";
2  char texte2[20]= "LES AMIS";
3  int n = strcmp(texte1,texte2); // n est positif
```

Copie (string.h):

Nom : strcpy

Prototype **: void *strcpy(char *chaine1,char *chaine2);**

Fonctionnement : recopie chaine2 dans chaine1 et renvoie l'adresse de chaîne1.

Exemple d'utilisation :

```
1  char texte2[20] = "BONJOUR ";
2  char texte1[20];
3  strcpy(texte1,texte2); // texte2 est inchangée, texte1 vaut maintenant
                                  "BONJOUR "
```

CHAPITRE 5

POINTEURS ET TABLEAUX

1. Introduction :

Les pointeurs et les tableaux sont deux des concepts de base en C++ qui permettent aux développeurs de travailler avec des données en mémoire.

Les pointeurs sont des variables spéciales qui stockent des adresses mémoire plutôt que des valeurs. Ils permettent aux développeurs d'accéder à des données en mémoire de manière flexible et efficace. Les pointeurs peuvent être utilisés pour manipuler les variables en passant simplement leur adresse mémoire, ce qui peut faciliter la création de fonctions plus complexes et l'allocation dynamique de mémoire.

Les tableaux sont des structures de données qui permettent de stocker plusieurs éléments de même type sous un même nom. Les tableaux sont souvent utilisés pour stocker des collections de données qui peuvent être traitées de manière similaire, telles que les scores d'un jeu ou les noms d'utilisateurs dans un système de gestion de bases de données. Les tableaux peuvent être accédés en utilisant un indice numérique, ce qui permet de travailler de manière efficace avec des collections de données de grande taille.

En C++, les pointeurs peuvent être utilisés pour accéder aux éléments d'un tableau, ce qui permet de manipuler des données en mémoire de manière flexible et efficace. Les pointeurs et les tableaux peuvent être utilisés ensemble pour résoudre de nombreux problèmes complexes, tels que la gestion de mémoire dynamique et la manipulation de données complexes

2. Les pointeurs :

Que ce soit dans la conduite de process, ou bien dans la programmation orientée objet, l'usage des pointeurs est extrêmement fréquent en C++.

2.1. L'operateur adresse & :

L'opérateur adresse & indique l'adresse d'une variable en mémoire.

Exemple : int i = 8;

```
cout<<"VOICI i:"<<i;
cout<<"\nVOICI SON ADRESSE EN HEXADECIMAL:"<<&i;
```

Définition : Un pointeur est une adresse. On dit que le pointeur pointe sur une variable dont le type est défini dans la déclaration du pointeur. Il contient l'adresse de la variable.

2.2. Déclaration Des Pointeurs :

Une variable de type pointeur se déclare à l'aide du type de l'objet pointé précédé du symbole *.

Exemple : char *pc ; pc est un pointeur sur un objet de type char

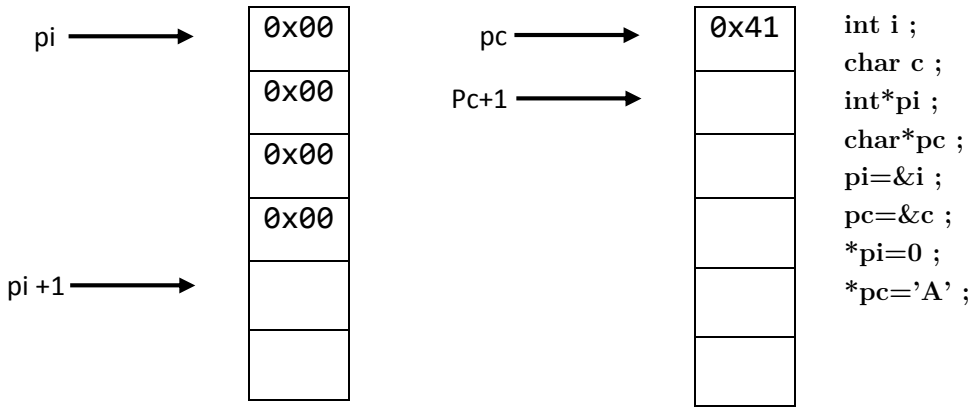
int *pi ; pi est un pointeur sur un objet de type int

float *Pr ; pr est un pointeur sur un objet de type float

2.3. Arithmétique Des Pointeurs :

On peut essentiellement déplacer un pointeur dans un plan mémoire à l'aide des opérateurs d'addition, de soustraction, d'incrémentation, de décrémentation. On ne

peut le déplacer que d'un nombre de cases mémoire multiple du nombre de cases réservées en mémoire pour la variable sur laquelle il pointe.



Exemples :

int *pi; // pi pointe sur un objet de type entier

float *pr; // pr pointe sur un objet de type réel

char *pc; // pc pointe sur un objet de type caractère

*pi = 421; // 421 est le contenu de la case mémoire p et des 3 suivantes

*(pi+1) = 53; // on range 53, 4 cases mémoire plus loin

*(pi+2) = 0xabcd; // on range 0xabcd 8 cases mémoire plus loin

*pr = 45.7; // 45,7 est rangé dans la case mémoire r et les 3 suivantes

*pc = 'j'; // le contenu de la case mémoire **c** est le code ASCII de 'j'

2.4. Affectation d'une valeur a un pointeur et allocation dynamique :

Lorsque l'on déclare une variable char, int, float .... un nombre de cases mémoire bien défini est réservé pour cette variable. En ce qui concerne les pointeurs, l'allocation de la case mémoire pointée obéit à des règles particulières :

Exemple :

char *pc;

Si on se contente de cette déclaration, le pointeur pointe « n'importe où ». Son usage, tel que, dans un programme peut conduire à un « plantage » du programme ou du système d'exploitation si les mécanismes de protection ne sont pas assez robustes.

L'initialisation du pointeur n'ayant pas été faite, on risque d'utiliser des adresses non autorisées ou de modifier d'autres variables.

3. Les tableaux :

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

Les tableaux correspondent aux vecteurs et matrices en mathématiques. Un tableau est caractérisé par sa taille et par le type de ses éléments.

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.

3.1. Les tableaux à une dimension :

Déclaration : type nom [dim]; **Exemples:** `int compteur[10];`

`float nombre [20] ;`

Cette déclaration signifie que le compilateur réserve dim places en mémoire pour ranger les éléments du tableau.

Exemples :

`int compteur [10]` ; le compilateur réserve des places en mémoire pour 10 entiers, soit 40 octets.

`float nombre [20]` ; le compilateur réserve des places en mémoire pour 20 réels, soit 80 octets.

Exemples :

```
1  #include <iostream.h>
2  #include <conio.h>
3  void main()
4  {
5  compteur[2] = 5;
6  nombre[i] = 6.789;
7  cout<<compteur[i];
8  cin>>nombre[i];
9  getch() ;
10 }
```

Il n'est pas nécessaire de définir tous les éléments d'un tableau. Toutefois, les valeurs non initialisées contiennent alors des valeurs quelconques.

3.2. Les tableaux à plusieurs dimensions :

Tableaux à deux dimensions :

Déclaration : type nom [dim1][dim2] ; **Exemples:**
```
int compteur[4][5];
float nombre[2][10];
```

Utilisation : Un élément du tableau est repéré par ses indices. En langage C++ les tableaux commencent aux indices 0. Les indices maxima sont donc dim1-1, dim2-1.

Exemples :

```
1  #include <iostream.h>
2  #include <conio.h>
3  void main()
4  {
5  compteur[2][4] = 5;
6  nombre[i][j] = 6.789;
7  cout<<compteur[i][j];
8  cin>>nombre[i][j];
9  getch() ;
10 }
```

Il n'est pas nécessaire de définir tous les éléments d'un tableau. Les valeurs non initialisées contiennent alors des valeurs quelconques.

3.3. Initialisation des tableaux :

On peut initialiser les tableaux au moment de leur déclaration :

Exemples :

```
int liste[10] = {1,2,4,8,16,32,64,128,256,528};
float nombre[4] = {2.67,5.98,-8.0,0.09};
int x[2][3] = {{1,5,7},{8,4,3}}; // 2 lignes et 3 colonnes
```

4. Tableaux et pointeurs :

En déclarant un tableau, on dispose d'un pointeur (adresse du premier élément du tableau).

Le nom d'un tableau est un pointeur sur le premier élément.

- **Les tableaux à une dimension :**

Les écritures suivantes sont équivalentes :

/*Allocation dynamique pendant l'exécution du programme */ int *tableau=new int[10] ;	/*Allocation automatique pendant la compilation du programme */ int tableau [10] ;	**Déclaration**
*tableau	**tableau [0]**	**Le 1er élément**
*(tableau+i)	**tableau [i]**	**Un élément d'indice i**
Tableau	**&tableau [0]**	**Adresse du 1er élément**
(tableau+i)	**&(tableau [i])**	**Adresse d'un élément i**

Table 6. Déclaration d'allocation dynamique et automatique des tableaux **à** une dimension

Il en va de même avec un tableau de réels (float).

Remarque :

- La déclaration d'un tableau entraîne automatiquement la réservation de places en mémoire. C'est aussi le cas si on utilise un pointeur et l'allocation dynamique en exploitant l'opérateur new.

- On ne peut pas libérer la place réservée en mémoire pour un tableau créé en allocation automatique (la réservation étant réalisée dans la phase de compilation en dehors de l'exécution). Par contre, en exploitant l'allocation dynamique via un pointeur, la primitive **delete** libère la mémoire allouée.

- **Les tableaux à plusieurs dimensions :**

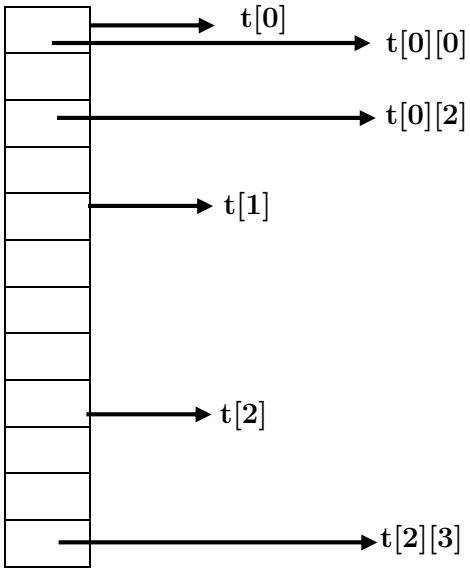
Un tableau à plusieurs dimensions est un pointeur de pointeur.

Exemple : `int t[3][4];` t est un pointeur de 3 tableaux de 4 éléments ou bien de 3 lignes à 4 éléments.

Les écritures suivantes sont équivalentes :

t[0]	**&t[0][0]**	adresse du 1er élément
t[1]	**&t[1][0]**	adresse du 1er élément de la 2eme ligne
t[i]	**&t[i][0]**	adresse du 1er élément de la ième ligne

t[i] +1 **&(t[i][0])+1** adresse du 1er élément de la i^{ème} +1 ligne



Répartition des éléments en mémoire

Le tableau a été déclaré ainsi int t[3][4]

CHAPITRE 6

LES FONCTIONS

1. Introduction :

Nos programmes deviennent de plus en plus complets, nous sommes en mesure de faire de plus en plus de choses. Mais, chaque fois que nous avons besoin de réutiliser un morceau de code, comme dans le cadre de la protection des entrées, nous nous rendons comptes que nous devons **dupliquer** notre code. Autant dire que c'est **contraignant**, ça **alourdit le code** et c'est une **source d'erreurs potentielles**.

Une fonction possède un et un seul point d'entrée, mais éventuellement plusieurs points de sortie (à l'aide du mot réservé **return – retourner dans le programme appelant**).

Le programme principal (**void main()**), est une fonction parmi les autres. C'est le point d'entrée du programme obligatoire et unique.

Une variable connue uniquement d'une fonction ou est une variable locale.

Une variable connue de tous les programmes est une variable globale.

2. Fonctions sans passage d'arguments et ne renvoyant rien au programme :

Elle est exécutée, mais le programme appelant ne reçoit aucune valeur de retour.

Exemple à expérimenter :

```
1  #include <iostream.h>
2  #include <conio.h>
3  void bonjour() // déclaration de la fonction
4  {
5  cout<<"bonjour\n";
6  }
7  void main() // programme principal
8  {
9  // appel de la fonction, exécution à partir de sa 1ere ligne
10 bonjour();
11 cout<<"POUR CONTINUER FRAPPER UNE TOUCHE : ";
12 getch();
13 }
```

Note

Il ne faut pas confondre déclaration avec exécution.

Les fonctions sont déclarées au début du fichier source. Mais elles ne sont exécutées que si elles sont appelées par le programme principal ou le sous-programme.

Une fonction peut donc être décrite en début de programme mais ne jamais être exécutée.

L'expression void bonjour() est appelé le prototype de la fonction " bonjour ".

Exemple à expérimenter :

```
1  #include <iostream.h>
2  #include <conio.h>
3  void bonjour() // déclaration de la fonction
4  {
5  cout<<"bonjour\n";
6  }
7  void coucou() // déclaration de la fonction
8  {
9  bonjour(); // appel d'une fonction dans une fonction
10 cout<<"coucou\n";
11 }
12 void main() // programme principal
13 {
14 coucou(); // appel de la fonction
15 cout<<"POUR CONTINUER FRAPPER UNE TOUCHE : ";
16 getch();
17 }
```

Note

Un programme peut contenir autant de fonctions que nécessaire.

Une fonction peut appeler une autre fonction. Cette dernière doit être déclarée avant celle qui l'appelle.

Par contre, l'imbrication de fonctions n'est pas autorisée en C++, une fonction ne peut pas être déclarée à l'intérieur d'une autre fonction.

L'expression void coucou () est appelé le prototype de la fonction " coucou "

Exemple à expérimenter :

```
1  #include <iostream.h>
2  #include <conio.h>
3  void carre() // déclaration de la fonction
4  {
5    int n, n2; // variables locales à carre
6    cout<<"ENTRER UN NOMBRE : ";
7    cin>>n;
8    n2 = n*n;
9    cout<<"VOICI SON CARRE : "<<n2<<"\n";
10 }
11 void main() // programme principal
12 {
13   clrscr();
14   carre(); // appel de la fonction
15   cout<<"POUR CONTINUER FRAPPER UNE TOUCHE : ";
16   getch();
17 }
```

Note

Les variables n et n2 ne sont connues que de la fonction carre. Elles sont appelées variables locales à carre. Aucune donnée n'est échangée entre main() et la fonction.

L'expression void carre() est le prototype de la fonction " carre ".

Exemple à expérimenter :

```
1  #include <iostream.h>
2  #include <conio.h>
3  void carre() // déclaration de la fonction
4  {
5     int n, n2; // variables locales à carre
6     cout<<"ENTRER UN NOMBRE : "; cin>>n;
7     n2 = n*n;
8     cout<<"VOICI SON CARRE : "<<n2<<"\n";
9  }
10 void cube() // déclaration de la fonction
11 {
12    int n, n3; // variables locales à cube
13    cout<<"ENTRER UN NOMBRE : "; cin>>n;
14    n3 = n*n*n;
15    cout<<"VOICI SON CUBE : "<<n3<<"\n";
16 }
17 void main() // programme principal
18 {
19    char choix; // variable locale à main()
20    cout<<"CALCUL DU CARRE TAPER 2\n";
21    cout<<"CALCUL DU CUBE TAPER 3\n";
22    cout<<"\nVOTRE CHOIX : "; cin>>choix;
23    switch(choix)
24       {
25         ♦ case '2':carre();
26          ← break;
27         ♦ case '3':cube();
28          ← break;
29        }
30    cout<<"\nPOUR CONTINUER FRAPPER UNE TOUCHE: ";
31    getch();
32 }
```

Note

Les 2 variables locales n sont indépendantes l'une de l'autre.

La variable locale choix n'est connue que de main().

Aucune donnée n'est échangée entre les fonctions et le programme principal.

L'expression void cube() est le prototype de la fonction " cube "

Exemple à expérimenter :

```
1  #include <iostream.h>
2  #include <conio.h>
3  int n; // variable globale, connue de tous les programmes
4  void carre() // déclaration de la fonction
5  {
6    int n2; // variable locale à carre
7    cout<<"ENTRER UN NOMBRE: "; cin>>n;
8    n2 = n*n;
9    cout<<"VOICI SON CARRE: "<<n2<<"\n";
10 }
11 void cube() // déclaration de la fonction
12 {
13    int n3; // variable locale à cube
14    cout<<"ENTRER UN NOMBRE : "; cin>>n;
15    n3 = n*n*n;
16    cout<<"VOICI SON CUBE: "<<n3<<"\n";}
17 void main() // programme principal
18 {
19    char choix; // variable locale à main()
20    cout<<"CALCUL DU CARRE TAPER 2\n";
21    cout<<"CALCUL DU CUBE TAPER 3\n";
22    cout<<"\nVOTRE CHOIX : "; cin>>choix;
23    switch(choix)
24       {
25         ♦ case '2':carre();
26          ← break;
27         ♦ case '3':cube();
28          ← break;
29        }
30    cout<<"\nPOUR CONTINUER FRAPPER UNE TOUCHE: ";
31    getch();
32 }
33
```

Note

La variable globale n est connue de tous les programmes (fonctions et programme principal)

La variable locale choix n'est connue que du programme principal main.

L'échange d'information entre la fonction et le programme principal se fait via cette variable globale.

Un programme bien construit (lisible, organisé par fonctions, et donc maintenable) doit posséder un minimum de variables globales.

3. Fonction renvoyant une valeur au programme :

3.1. Sans passage d'arguments :

Dans ce cas, la fonction, après exécution, renvoie une valeur. Le type de cette valeur est déclaré avec la fonction. La valeur retournée est spécifiée à l'aide du mot réservé return.

Cette valeur peut alors être exploitée par le sous-programme appelant.

Le transfert d'information a donc lieu de la fonction vers le sous-programme appelant.

Exemple à expérimenter :

```
1  #include <iostream.h>
2  #include <conio.h>
3  #include <stdlib.h>
4  #include <time.h>
5  int lance_de() // déclaration de la fonction
6  { // random(n) retourne une valeur comprise entre 0 et n-1
7  int test = random(6) + 1; //variable locale
8  return test;
9  }
10 void main()
11 {
12   int resultat;
13   randomize();
14   resultat = lance_de();
15   // résultat prend la valeur retournée par le sous-programme
16   cout<<"Vous avez obtenu le nombre: "<<résultat<<"\n";
17   cout<<"POUR SORTIR FRAPPER UNE TOUCHE ";
18   getch();
19 }
```

3.2. Fonctions avec passage d'arguments :

Ces fonctions utilisent les valeurs de certaines variables du sous-programme les ayant appelés : on passe ces valeurs au moyen d'arguments déclarés avec la fonction.

Le transfert d'information a donc lieu du programme appelant vers la fonction.

Ces fonctions peuvent aussi, si nécessaire, retourner une valeur au sous-programme appelant via le mot réservé return.

Exemple à expérimenter :

```
1  #include <iostream.h>
2  #include <conio.h>
3  int carre(int x) // déclaration de la fonction
4  {  // x est un paramètre formel
5  int x2 = x*x; // variable locale
6  return x2 ;
7  }
8  void main()
9  {
10 int n1, n2, res1, res2; /* variables locales au main */
11 cout<<"ENTRER UN NOMBRE : "; cin>>n1;
12 res1 = carre(n1);
13 cout<<"ENTRER UN AUTRE NOMBRE : ";
14 cin>>n2; res2 = carre(n2);
15 cout<<"VOICI LEURS CARRES : "<<res1<<" "<< res2;
16 cout<<"\n\nPOUR SORTIR FRAPPER UNE TOUCHE : ";
17 getch();
18 }
```

Note

x est un paramètre formel, ou argument. Ce n'est pas une variable effective du programme.

Sa valeur est donnée via n1 puis n2 par le programme principal.

On dit que l'on a passé le paramètre PAR VALEUR.

On peut ainsi appeler la fonction carre autant de fois que l'on veut avec des variables différentes.

Il peut y avoir autant de paramètre que nécessaire. Ils sont alors séparés par des virgules.

S'il y a plusieurs arguments à passer, il faut respecter la syntaxe suivante :

void fonction1 (int x, int y) void fonction2(int a, float b, char c)

4. Passage des tableaux et des pointeurs en argument :

Exemple à expérimenter :

```
#include <iostream.h>
#include <conio.h>
int somme(int *tx, int taille)
{
int total=0;
      for(int i=0;i<taille;i++)
            total = total + tx[i];
      return total;
}
void main()
{
int tab[20]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19};
int resultat = somme(tab, 20);
cout<<"Somme des éléments du tableau : "<<resultat;
cout<<"\nPOUR CONTINUER FRAPPER UNE TOUCHE ";
getch();
}
```

Note

Le tableau est passé au sous-programme comme un pointeur sur son premier élément. On dit, dans ce cas, que l'on a passé le paramètre PAR ADRESSE. Le langage C++, autorise, dans une certaine mesure, le non-respect du type des arguments lors d'un appel à fonction : le compilateur opère alors une conversion de type.

5. Surcharge d'une fonction :

La surcharge de fonctions est définie comme le processus consistant à avoir deux fonctions ou plus portant le même nom, mais de paramètres différents.

Dans la surcharge de fonction, la fonction est redéfinie en utilisant différents types d'arguments ou un nombre différent d'arguments. Ce n'est que par ces différences que le compilateur peut différencier les fonctions.

L'avantage de la surcharge de fonctions est qu'elle augmente la lisibilité du programme car il n'est pas nécessaire d'utiliser des noms différents pour la même action.

Exemple à expérimenter :

```
1  #include <iostream.h>
2  #include <conio.h>
3  void test(int n = 0, float x = 2.5)
4  {
5    cout <<"Fonction n°1 : ";
6    cout << "n= "<<n<<" x="<<x<<"\n";
7  }
8  void test(float x = 4.1,int n = 2)
9  {
10   cout <<"Fonction n°2 : ";
11   cout << "n= "<<n<<" x="<<x<<"\n";
12 }
13 void main()
14 {
15   int i = 5; float r = 3.2;
16   test(i,r); // fonction n°1
17   test(r,i); // fonction n°2
18   test(i); // fonction n°1
19   test(r); // fonction n°2
20  // les appels suivants, ambigus, sont rejetés par le compilateur
21  // test();
22  // test (i,i);
23  // test (r,r);
24  // les initialisations par défaut de x à la valeur 4.1
25  //et de n à 0 sont inutilisables
26 getch();
27 }
```

CHAPITRE 7

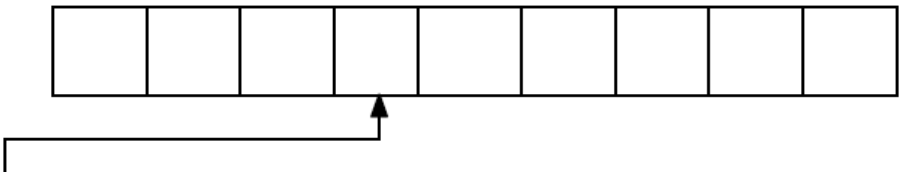
LES FICHIERS

1. Introduction :

Un fichier est un ensemble d'informations stockées sur une mémoire de masse (disque dur, disquette, bande magnétique, CD-ROM).

Ces informations sont sauvegardées à la suite les unes des autres et ne sont pas forcément de même type (**un char, un int, une structure** ...)

Un pointeur permet de se repérer dans le fichier. On accède à une information en amenant le pointeur sur sa position.



Pointeur

Sur le support de sauvegarde, le fichier possède un nom. Ce nom est composé de 2 parties : le nom proprement dit et l'extension. L'extension donne des informations sur le type d'informations stockées (à condition de respecter les extensions associées au type du fichier).

Exemples :

toto.txt Le fichier se nomme toto et contient du texte

mon_cv.doc	Le fichier se nomme mon_cv et contient du texte, il a été édité sou WORD
ex1.cpp	Le fichier se nomme ex1 et contient le texte d'un programme écrit en C++ (fichier source)
ex1.exe	Le fichier se nomme ex1, il est exécutable
bibi.dll	Le fichier se nomme bibi, c'est un fichier nécessaire à l'exécution d'un autre logiciel

On distingue deux façons de coder les informations stockées dans un fichier :

- **Ficher en ASCII (TEXTE) :**

  Fichier dit « texte », les informations sont codées en ASCII. Ces fichiers sont listables et éditables. Le dernier octet de ces fichiers est EOF (End Of File - caractère ASCII spécifique). Ils peuvent posséder les extensions .TXT, .DOC, .RTF, .CPP, .BAS, .PAS, .INI etc …

- **Ficher en binaire :**

  Fichier dit « binaire », les informations sont codées telles que. Ce sont en général des fichiers de nombres. Ils ne sont ni listables, ni éditables. Ils possèdent par exemple les extensions **.OBJ, .BIN, .EXE, .DLL, .PIF** etc …

2. Les Opérations possibles avec les fichiers :

- Créer
- Ouvrir
- Lire
- Ecrire
- Détruire
- Renommer
- Fermer.

La plupart des fonctions permettant la manipulation des fichiers sont rangées dans la bibliothèque standard **stdio.h**, certaines dans la bibliothèque **io.h** pour le **BORLAND C++**.

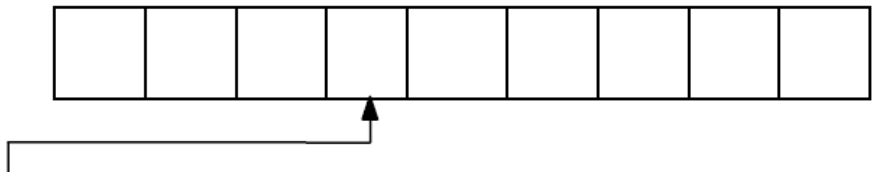
Pour manipuler un fichier, on commence toujours par l'ouvrir et vérifier qu'il est effectivement ouvert (s'il n'existe pas, cela correspond à une création).

Lorsque la manipulation est terminée, il faut fermer ce fichier et vérifier sa fermeture effective.

Le langage C++ ne distingue pas les fichiers à accès séquentiel des fichiers à accès direct, certaines fonctions de la bibliothèque livrée avec le compilateur permettent l'accès direct. Les fonctions standards sont des fonctions d'accès séquentiel.

2.1. Déclaration : `FILE *index ;` `// majuscules obligatoires pour FILE`

On définit un pointeur. Il s'agit du pointeur représenté sur la figure du début de chapitre. Ce pointeur repère une cellule donnée.



Index

index est la variable qui permettra de manipuler le fichier dans le programme.

2.2. Ouverture :

Il faut associer à la variable index au nom du fichier sur le support. On utilise la fonction fopen de prototype `FILE *fopen (char *nom, char *mode) ;`

On passe donc 2 chaînes de caractères

nom : celui figurant sur le support, par exemple : « a : \toto.dat »

- **mode (pour les fichiers TEXTES) :**

  « r » lecture seule

  « w » écriture seule (destruction de l'ancienne version si elle existe)

  « w+ » lecture/écriture (destruction ancienne version si elle existe)

« r+ » lecture/écriture d'un fichier existant (mise à jour), pas de création d'une nouvelle version.

« a+ » lecture/écriture d'un fichier existant (mise à jour), pas de création d'une nouvelle version, le pointeur est positionné à la fin du fichier.

- **mode (pour les fichiers BINAIRES) :**

« rb » lecture seule

« wb » écriture seule (destruction de l'ancienne version si elle existe)

« wb+ »lecture/écriture (destruction ancienne version si elle existe)

« rb+ » lecture/écriture d'un fichier existant (mise à jour), pas de création d'une nouvelle version.

« ab+ »lecture/écriture d'un fichier existant (mise à jour), pas de création d'une nouvelle version, le pointeur est positionné à la fin du fichier.

A l'ouverture, le pointeur est positionné au début du fichier (sauf « a+ » et « ab+ », à la fin).

Exemple1 :
```
FILE *index ;
index = fopen("a :\\toto.dat", "rb") ;
```

Exemple2 :
```
FILE *index ;
char nom[30] ;
cout<< "Nom du fichier : " ;
cin >> nom ;
index = fopen(nom, "w") ;
```

2.3. Fermeture :

On utilise la fonction de prototype `int fclose(FILE *f);`

Cette fonction retourne 0 si la fermeture s'est bien passée.

Exemple :
```
FILE *index ;
index = fopen("a :\\toto.dat", "rb") ;
// Ici instructions de traitement
fclose(index) ;
```

2.4. Destruction :

On utilise la fonction de prototype int `remove(char *nom);`

Cette fonction retourne 0 si la fermeture s'est bien passée.

Exemple :
```
int x ;
x = remove("a :\\toto.dat") ;
if (x == 0)
cout << " Fermeture OK : " ;
else
cout << " Problème à la fermeture : " ;
```

2.5. Renommer :

On utilise la fonction de prototype `int rename (char *oldname, char *newname);`

Cette fonction retourne 0 si la fermeture s'est bien passée.

Exemple :
```
int x ;
x = rename("a :\\toto.dat", "a :\\tutu.dat") ;
if (x == 0)
cout << " Operation OK : " ;
else
cout << "L'operation s'est mal passee : " ;
```

3. Manipulations des fichiers :

3.1. Manipulations des fichiers textes :

3.1.1. Ecriture dans le fichier :

La fonction de prototype `int putc(char c, FILE *index)` écrit la valeur de c à la position courante du pointeur, le pointeur avance d'une case mémoire.

Cette fonction retourne –1 en cas d'erreur.

Exemple : putc('A', index) ;

La fonction de prototype `int fputs(char *chaîne, FILE *index)` est analogue avec une chaîne de caractères. Le pointeur avance de la longueur de la chaîne ('\0' n'est pas rangé dans le fichier).

Cette fonction retourne le code ASCII du caractère, retourne -1 en cas d'erreur (par exemple tentative d'écriture dans un fichier ouvert en lecture)

Exemple : `fputs("BONJOUR ! ", index) ;`

3.1.2. Relecture d'un fichier :

Les fichiers texte se terminent par le caractère ASCII EOF (de code –1). Pour relire un fichier, on peut donc exploiter une boucle jusqu'à ce que la fin du fichier soit atteinte.

- La fonction de prototype int getc(FILE *index) lit 1 caractère, et retourne son code ASCII, sous forme d'un entier. Cet entier vaut –1 (EOF) en cas d'erreur ou bien si la fin du fichier est atteinte. Via une conversion automatique de type, on obtient le caractère.
- La fonction de prototype `char *fgets(char *chaine, int n, FILE *index)` lit n-1 caractères à partir de la position du pointeur et les range dans chaîne en ajoutant '\0'.

  Retourne un pointeur sur la chaîne, retourne le pointeur NULL en cas d'erreur, ou bien si la fin du fichier est atteinte.

Exemple :

```
FILE *index ;
char texte[10] ;
// Ouverture
fgets(texte, 7, index) ; // lit 7 caractères dans le
fichier et forme la chaîne
// « texte » avec ces caractères
```

La fonction de prototype `int getw(FILE *index)` lit 1 nombre stocké sous forme ASCII dans le fichier, et le retourne. Cet entier vaut –1 en cas d'erreur ou bien si la fin du fichier est atteinte.

3.2. Manipulations des fichiers binaires

- La fonction `int feof(FILE *index)` retourne 0 tant que la fin du fichier n'est pas atteinte.
- La fonction `int ferror(FILE *index)` retourne 1 si une erreur est apparue lors d'une manipulation de fichier, 0 dans le cas contraire.
- La fonction de prototype `int fwrite(void *p, int taille_bloc, int nb_bloc, FILE *index)` écrit à partir de la position courante du pointeur index nb_bloc X taille_bloc octets lus à partir de l'adresse p. Le pointeur fichier avance d'autant.

  Le pointeur p est vu comme une adresse, son type est sans importance.

  Cette fonction retourne le nombre de blocs écrits (0 en cas d'erreur, ou bien si la fin du fichier est atteinte).

  Exemple :

  ```
  taille_bloc = 4 (taille d'un entier en C++), nb_bloc=3, écriture de 3 entiers.
  int tab[10] ;
  fwrite(tab,4,3,index) ;
  ```

- La fonction de prototype `int fread(void *p,int taille_bloc,int nb_bloc,FILE *index)` est analogue à fwrite en lecture. Cette fonction retourne le nombre de blocs luts (0 en cas d'erreur, ou bien si la fin du fichier est atteinte).

CHAPITRE 8

PROGRAMMATION ORIENTEE OBJET EN C++

1. Introduction :

On attend d'un programme informatique l'exactitude (réponse aux spécifications), la robustesse (réaction correcte à une utilisation « hors normes »), l'extensibilité (aptitude à l'évolution), la réutilisabilité (utilisation de modules), la portabilité (support d'une autre implémentation) et l'efficience (performance en termes de vitesse d'exécution et de consommation mémoire).

Le C++ et un langage orienté objet. Un langage orienté objet permet la manipulation de classes. Comme on le verra dans ce chapitre, la classe généralise la notion de structure. Une classe contient des variables (ou « données ») et des fonctions (ou « méthodes ») permettant de manipuler ces variables.

Les langages « orientés objet » ont été développés pour faciliter l'écriture et améliorer la qualité des logiciels en termes de modularité et surtout de réutilisation.

Un langage orienté objet est livré avec une bibliothèque de classes. Le développeur utilise ces classes pour mettre au point ses logiciels.

C'est là qu'intervient la programmation orientée objet (en abrégé P.O.O), fondée justement sur le concept d'objet, à savoir une association des données et des procédures (qu'on appelle alors méthodes) agissant sur ces données.

2. Notion d'objet de classe et d'encapsulation :

2.1 Notion d'objet :

Il est impossible de parler de Programmation Orientée Objet sans parler d'objet, bien entendu. Tâchons donc de donner une définition aussi complète que possible d'un objet. Un objet est avant tout une structure de données. Autrement dit, il s'agit d'une entité chargée de gérer des données, de les classer, et de les stocker sous une certaine forme. En cela, rien ne distingue un objet d'une quelconque autre structure de données. La principale différence vient du fait que l'objet regroupe les données et les moyens de traitement de ces données.

Un objet rassemble de fait deux éléments de la programmation procédurale.

- **Les champs :** Les champs sont à l'objet ce que les variables sont à un programme : ce sont eux qui ont en charge les données à gérer. Tout comme n'importe quelle autre variable, un champ peut posséder un type quelconque défini au préalable : nombre, caractère... ou même un type objet.
- **Les méthodes :** Les méthodes sont les éléments d'un objet qui servent d'interface entre les données et le programme. Sous ce nom obscur se cachent simplement des procédures ou fonctions destinées à traiter les données.

Les champs et les méthodes d'un objet sont ses membres. Si nous résumons, un objet est un ensemble de données sur lesquelles des procédures peuvent être appliquées. Ces procédures applicables au donnés sont appelées méthodes stocker des données dans des champs et à les gérer au travers des méthodes. La programmation d'un objet se fait

donc en indiquant les données de l'objet et en définissant les procédures qui peuvent lui être appliquées.

Pour appeler la méthode d'un objet, on utilise cette écriture :

objet.methode()

Exemple : méthodes utiles du type string

La méthode size() : permet de connaître la longueur de la chaîne actuellement stockée dans l'objet de type string. Cette méthode ne prend aucun paramètre et renvoie la longueur de la chaîne. Comme vous venez de le découvrir, il va falloir appeler la méthode de la manière suivante :

```
1  int main()
2  {
3  string maChaine("leila !");
4  cout << "Longueur de la chaine : " << maChaine.size();
5  return 0;
6  }
```

Exemple : Objet Cercle

```
1  {Rayon, abs, ord de type réel // 3 Champs de type réels
2  Méthode périmètre de type réel // fonction qui calcul le périmètre du
3  cercle
4  {
5  Périmètre= 2*3,14*rayon
6  }
7  Méthode surface de type réel // fonction qui calcul la surface du cercle
8  {
9  surface= 2*3,14*rayon*rayon
10 }
11 }
```

2.2 Notion de classe :

Avec la notion d'objet, il convient d'amener la notion de classe. Il s'agit donc du type à proprement parler. L'objet en lui-même est une instance de classe, plus simplement

un exemplaire d'une classe, sa représentation en mémoire. Par conséquent, on déclare comme type une classe, et on déclare des variables de ce type appelées des objets.

Une classe "Livre" par exemple rassemblant son titre, son auteur, son année de parution et son nombre de pages et peut une méthode nommée obtenir Information () qui retourne une chaîne contenant la valeur des champs de l'objet courant.

Une classe est la généralisation de la notion de type défini par l'utilisateur, dans lequel se trouvent associées à la fois des données (membres données) et des méthodes (fonctions membres).

En langage C++, La déclaration d'une classe est voisine de celle d'une structure. En effet, il suffit :

- De remplacer le mot clé **struct** par le mot clé **class.**
- De préciser quels sont les membres publics (fonctions ou données) et les membres privés en utilisant les mots clés **publics** et **private.**

Par conséquent, on déclare comme type une classe et on déclare des variables de ce type appelées objets.

Une classe définit donc la structure des données alors appelées champs ou variables instances, que les objets correspondants auront, ainsi que les méthodes de l'objet.

L'initialisation de l'objet nouvellement créé est faite par une méthode spéciale, le constructeur. Lorsque l'objet est détruit, une autre méthode est appelée : le destructeur.

L'utilisateur peut définir ses propres constructeurs et destructeurs d'objets si nécessaires.

Une classe est composée de deux parties :

- **Les champs ou attributs** (parfois appelés données membres) : il s'agit des données représentant l'état de l'objet.
- **Les méthodes** (parfois appelées fonctions membres) : il s'agit des opérations applicables aux objets.

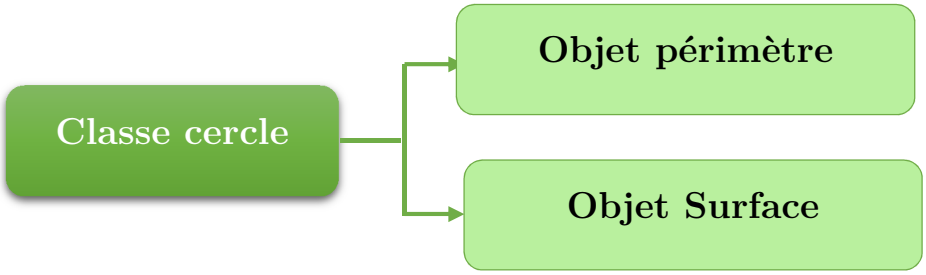


Figure 2. Classe cercle

Une classe est un module pour fabriquer des objets. Elle définit les méthodes, et elle décrit la structure des données.

Une classe est constituée :

- De variables, ici appelées attributs (on parle aussi de variables membres)
- De fonctions, ici appelées méthodes (on parle aussi de fonctions membres)

3. Fondamentaux de la POO :

La programmation " orientées objet " est dirigée par trois fondamentaux :

- Héritage
- Encapsulation
- Polymorphisme

3.1. Notion d'héritage :

L'héritage est un principe propre à la programmation orientée objet, permettant de créer une nouvelle classe à partir d'une classe existante. Le nom d'héritage (pouvant parfois être appelé dérivation de classe) provient du fait que la classe dérivée (la classe

nouvellement créée, ou classe fille) contient les attributs et les méthodes de sa classe mère (la classe dont elle dérive).

L'intérêt majeur de l'héritage est de pouvoir définir de nouveaux attributs et de nouvelles méthodes pour la classe dérivée, qui viennent s'ajouter à ceux et celles hérités.

Par ce moyen, une hiérarchie de classes de plus en plus spécialisées est créée. Cela a comme avantage majeur de ne pas avoir à repartir de zéro lorsque l'on veut spécialiser une classe existante.

De cette manière il est possible de récupérer des librairies de classes, qui constituent une base, pouvant être spécialisées à loisir. Une particularité de l'héritage est qu'un objet d'une classe dérivée est aussi du type de la classe mère.

Dans la classe fille, on trouve :

- De nouvelles méthodes et de nouveaux champs.
- Des méthodes qui surchargent celles de la classe mère càd redéfinissant.

Exemple :

Les classes voiture et moto décrivent de la classe Véhicule.

3.2. Notion D'encapsulation :

L'encapsulation est l'un des fondements de la POO. Elle permet d'utiliser un objet en cachant son fonctionnement interne. Grâce à l'encapsulation, les données privées ne sont pas accessibles de façon directe. Ceci permet de sécuriser les données qui ne peuvent pas être altérées par un utilisateur extérieur.

Les utilisateurs de l'objet disposent d'une interface public pour utiliser l'objet. L'accès aux données privées se fait alors uniquement par les fonctions membres de la classe. Un autre avantage de l'encapsulation est de rendre l'implémentation de la classe transparente aux utilisateurs. Tant que l'interface public de la classe n'a pas changée,

les utilisateurs peuvent continuer à utiliser la classe de la même façon même si l'implémentation de ses données a changée. Grâce à l'encapsulation, on peut changer la façon dont les données sont déclarées et stockées sans changer la façon dont elles sont utilisées.

3.3. Notion de polymorphisme :

Le polymorphisme signifie par définition qu'un objet (classe) peut prendre plusieurs formes. Suivant ce principe, on perçoit l'objet différemment selon sa composition ou sa relation avec un autre objet (classe).

Les objets sont dits polymorphes car ils possèdent plusieurs types : le type de leurs classes et les types des classes ascendantes.

Par exemple, si on prend la classe Véhicule, l'objet Honda, instance de Moto aura comme type initial Moto mais une Moto possède par héritage le type Véhicule. L'objet Honda est bien un véhicule.

4. Méthodes de la POO :

La programmation Orientée Objet utilise quelques méthodes qui sont presque communes à tous les langages de programmation objet tels que C++, JAVA et PHP nous décrivons ici deux méthodes particulières.

- Les constructeurs.
- Les destructeurs.

4.1. Les constructeurs :

Comme leur nom l'indique, les constructeurs servent à construire l'objet en mémoire.

Un constructeur va se charger de mettre en place les données, d'associer les méthodes avec les attributs et de créer le diagramme d'héritage de l'objet, autrement dit de

mettre en place toutes les liaisons entre les ancêtres et descendants. Il peut exister en mémoire plusieurs instances d'un même type objet, par contre seule une copie des méthodes est conservée en mémoire, de sorte que chaque instance se réfère à la même zone mémoire en ce qui concerne les méthodes. Bien étendues attributs sont distincts d'un objet à un autre.

Un objet peut ne pas avoir de constructeur explicite, il est alors créé par le compilateur. Certains langages (comme le JAVA) autorisent d'avoir plusieurs constructeurs : c'est l'utilisateur qui décidera du constructeur à appeler. Comme pour toute méthode, un constructeur peut être surchargé, et donc effectuer diverses actions en plus de la construction même de l'objet. On utilise ainsi généralement les constructeurs pour initialiser les attributs de l'objet.

4.2. Les destructeurs :

Le destructeur se charge de déduire l'instance de l'objet. La mémoire allouée pour le diagramme d'héritage est libérée. Certains compilateurs peuvent également se servir des destructeurs pour éliminer de la mémoire le code correspondant aux méthodes d'un type d'objet si plus aucune de cet objet ne réside en mémoire. Tout comme pour les constructeurs, on objet peut ne pas avoir de destructeur. Une fois encore, c'est le compilateur qui se chargera de la destruction statique de l'objet.

Certains langages autorisant d'avoir plusieurs destructeurs, leur rôle commun reste identique, mais peut s'y ajouter la destruction de certaines variables internes pouvant différer d'un destructeur à l'autre. La plupart du temps, à un constructeur distinct est associé un destructeur distinct.

Un constructeur sert à créer une instance de l'objet càd de fabriquer un exemplaire de l'objet en mémoire ; le destructeur de charge de le détruire.

5. Programmation procédurale et programmation orientée objet :

La différence entre la programmation procédurale et la programmation orientée objet (POO) réside dans le fait que dans la programmation procédurale, les programmes sont basés sur des fonctions, et les données peuvent être facilement accessibles et modifiables, alors qu'en programmation orientée objet, chaque programme est constitué d'entités appelées objets, qui ne sont pas facilement accessibles et modifiables.

5.1. La programmation procédurale :

Dans la programmation procédurale le programme est divisé en petites parties appelées procédures ou fonctions. Comme son nom l'indique, la programmation procédurale contient une procédure étape par étape à exécuter. Ici, les problèmes sont décomposés en petites parties et ensuite, pour résoudre chaque partie, une ou plusieurs fonctions sont utilisées.

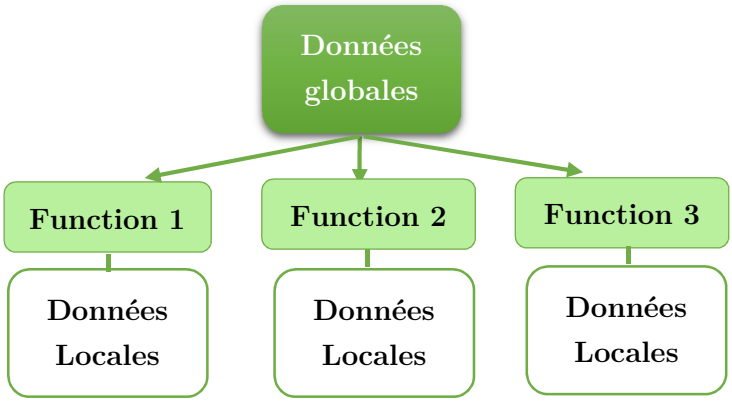


Figure 3. La programmation procédurale

5.2. La Programmation Orientée Objet (POO) :

Dans la programmation orientée objet le programme est divisé en parties appelées objets. La programmation orientée objet est un concept de programmation qui se concentre sur l'objet plutôt que sur les actions et les données plutôt que sur la logique.

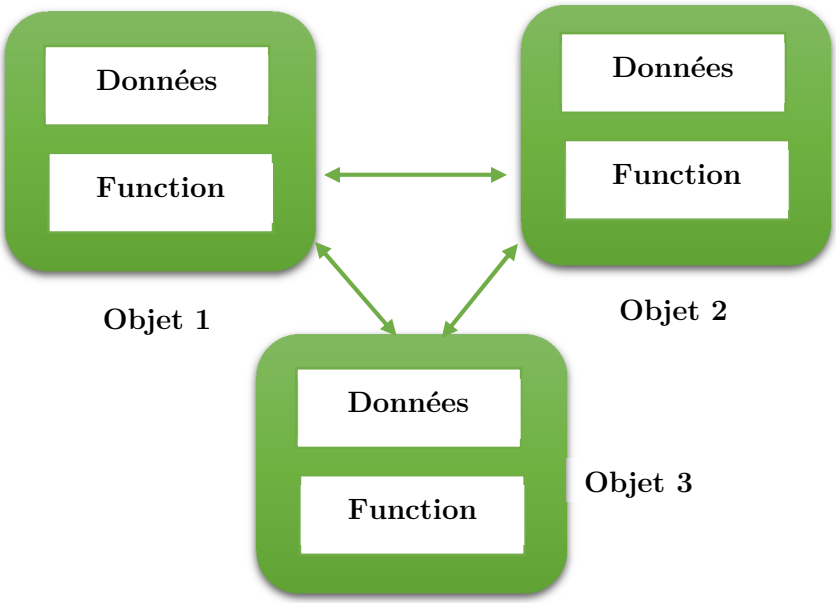


Figure 4. La programmation Orientée Objet

5.3. Table de comparaison :

	Programmation Procédurale	Programmation Orientée Objet
Programmes	Le programme principal est divisé en petites parties selon les fonctions.	Le programme principal est divisé en petit objet en fonction du problème.
Les données	Chaque fonction contient des données différentes.	Les données et les fonctions de chaque objet individuel agissent comme une seule unité.
Permission	Pour ajouter de nouvelles données au programme, l'utilisateur doit s'assurer que la fonction le permet.	Le passage de message garantit l'autorisation d'accéder au membre d'un objet à partir d'un autre objet.
Exemples	Pascal, Fortran	PHP5, C ++, Java.
Accès	Aucun spécificateur d'accès n'est utilisé.	Les spécificateurs d'accès public, private, et protected sont utilisés.
La communication	Les fonctions communiquent avec d'autres fonctions en gardant les règles habituelles.	Un objet communique entre eux via des messages.
Contrôle des données	La plupart des fonctions utilisent des données globales.	Chaque objet contrôle ses propres données.
Importance	Les fonctions ou les algorithmes ont plus d'importance que les données dans le programme.	Les données prennent plus d'importance que les fonctions du programme.

Table 7. Comparaison entre programmation procédurale et orientée objet

Note

Les failles de la programmation procédurale posent le besoin de la programmation orientée objet. La programmation orientée objet corrige les défauts de la programmation procédurale en introduisant le concept «objet» et «classe». Il améliore la sécurité des données, ainsi que l'initialisation et le nettoyage automatiques des objets. La programmation orientée objet permet de créer plusieurs instances de l'objet sans aucune interférence.

Conclusion Générale

Le C++ est un langage de programmation populaire et puissant utilisé pour développer des applications de divers types, allant des petits scripts simples à des systèmes complexes. Les fondamentaux de la programmation en C++ comprennent la compréhension de la syntaxe de base, des structures de contrôle de flux, des variables, des fonctions, des tableaux, des opérateurs et des pointeurs. La programmation orientée objet est également un élément clé du C++, ce qui en fait un outil puissant pour les développeurs.

Il est important de pratiquer régulièrement pour améliorer ses compétences en programmation en C++, ainsi que de se tenir au courant des mises à jour et des améliorations apportées au langage. Il est également utile de consulter la documentation et de participer à des communautés en ligne pour discuter de problèmes et de solutions avec d'autres développeurs.

En conclusion, les fondamentaux de la programmation en C++ sont une base solide pour développer des compétences en programmation et en développement de logiciels. Avec de la pratique et de la persévérance, vous pouvez devenir un développeur C++ expérimenté capable de créer des applications performantes et fiables

Travaux pratiques

Travaux pratiques (TP°1)

Exercice 01 :

Affichage d'une variable de type int ou float:
Tester le programme suivant et conclure.

```
#include <iostream.h>
#include <conio.h>
void main( )
{
int u = 1000 ;
float s = 45.78 ;
cout <<"Voici u (en base 10) : " << u << "\n";
cout <<"Voici u (en hexa) : "<< hex << u <<"\n";
cout <<"Voici s : "<< s << "\n";
cout <<"Pour continuer frapper une touche...";
getch( ); // Attente d'une saisie clavier
}
```

Exercice 02 :

a et b sont des entiers, a = -21430 b = 4782, calculer et afficher a+b, a-b, a*b, a/b, a%b en soignant l'interface homme/machine.
Indication: a/b donne le quotient de la division, a%b donne le reste de la division.

Exercice 03 : Affichage d'une variable de type char : tester le programme ci-dessous et conclure.

```
#include <iostream.h>
#include <conio.h>
void main()
{
char u,v,w;
int i;
u = 'A';
v = 67;
w = 0x45;
cout<<"Voici u : "<< u << "\n";
```

```
cout<<"Voici v : "<< v <<"\n";
cout<<"Voici w : "<< w <<"\n";
i = u; // conversion automatique de type
 // pour obtenir le code ascii de la lettre A en base 10
cout<<"Voici i : "<< i << "\n";
// pour obtenir le code ascii de la lettre A en hexadécimal
cout<<"Voici i : "<< hex << i << "\n";
cout<<"Pour continuer frapper une touche...";
getch(); // Attente d'une saisie clavier
}
```

Exercice 4:

Quels nombres va renvoyer le programme suivant ?

```
#include <iostream.h>
#include <conio.h>
void main ( )
{
cout<<"TAILLE D'UN CARACTERE : "<<sizeof(char)<< "\n";
cout<<"TAILLE D'UN ENTIER : " <<sizeof(int)<< "\n";
cout<<"TAILLE D'UN REEL : " <<sizeof(float)<< "\n";
cout<<"TAILLE D'UN DOUBLE : " <<sizeof(double)<< "\n";
cout <<"Pour continuer frapper une touche...";
getch( ); // Attente d'une saisie clavier
}
```

Travaux pratiques (TP°2)

Exercice 01 :

Dans une élection, I est le nombre d'inscrits, V le nombre de votants, Q le quorum, P = 100V/I le pourcentage de votants, M = V/2 + 1 le nombre de voix pour obtenir la majorité absolue. Le quorum est le nombre minimum de votants pour que le vote soit déclaré valable.

Ecrire un programme qui

1- demande à l'utilisateur de saisir I, Q et V,

2- teste si le quorum est atteint,

3- si oui calcule et affiche P, M, sinon affiche un message d'avertissement.

Exercice 02 :

Ecrire un programme permettant de saisir un entier n, de calculer n!, puis de l'afficher. Utiliser une boucle do ...while puis while puis for.

Exercice 03 :

La formule récurrente ci-dessous permet de calculer la racine du nombre 2 :

U0 = 1

Ui = (Ui-1+ 2/Ui-1) / 2

Ecrire un programme qui saisit le nombre d'itérations, puis calcule et affiche la racine de 2.

Exercice 04 :

Tester ce programme

```
#include<iostream.h>
#include<conio.h>
void main()
{
int a;
int *x,*y;
a=90;
x=&a;
cout << *x << "\n";
*x=100;
cout << "a vaut : " << a << "\n";
y=x;
*y=80;
cout << "a vaut : " << a << "\n";
getch();
}
```

Exercice 05 :

Soit le petit programme suivant :

```
#include <iostream.h>
#include <conio.h>
Void main()
{
int i, n, som ;
som = 0 ;
for (i=0 ; i<4 ; i++)
{ cout << "donnez un entier " ;
cin >> n ;
som += n ;
} cout << "Somme :"<< som ;
}
```

Écrire un programme réalisant exactement la même chose, en employant, à la place de l'instruction for : a. une instruction while, b. une instruction do ... while.

Travaux pratiques (TP°3)

Exercice 01 :

Écrire un programme qui lit 10 nombres entiers dans un tableau avant d'en rechercher le plus grand et le plus petit en utilisant uniquement le « formalisme tableau » ;

Exercice 02 :

Soient deux tableaux t1 et t2 déclarés ainsi :

float t1[10], t2[10] ;

Écrire les instructions permettant de recopier, dans t1, tous les éléments positifs de t2, en complétant éventuellement t1 par des zéros. Ici, on ne cherchera pas à fournir un programme complet et on utilisera systématiquement « formalisme usuel des tableaux» ;

Exercice 03 :

Saisir une matrice d'entiers 2x2, calculer et afficher son déterminant.

Exercice 04 :

Un programme contient la déclaration suivante :

int tab[10] = {4,12,53,19,11,60,24,12,89,19};

Compléter ce programme de sorte d'afficher les adresses des éléments du tableau

Exercice 05 :

Un programme contient la déclaration suivante :

int tab[20] = {4,-2,-23,4,34,-67,8,9,-10,11, 4,12,-53,19,11,-60,24,12,89,19};

Compléter ce programme de sorte d'afficher les éléments du tableau avec la présentation suivante :

4	-2	-23	4	34
-67	8	9	-10	11
4	12	-53	19	11
-60	24	12	89	19

Travaux pratiques (TP°4)

Exercice 1: Tester ce programme

```
#include <iostream.h>
#include <conio.h>
void carre( ) // déclaration de la fonction
{
int n, n2; // variables locales à carre
cout<<"ENTRER UN NOMBRE : "; cin>>n;
n2 = n*n;
cout<<"VOICI SON CARRE : "<<n2<<"\n";
}
void main() // programme principal
{
carre(); // appel de la fonction
getch();
}
```

Exercice 2 :

```
#include <iostream.h>
#include <conio.h>
void carre() // déclaration de la fonction
{
int n, n2; // variables locales à carre
cout<<"ENTRER UN NOMBRE : "; cin>>n;
n2 = n*n;
cout<<"VOICI SON CARRE : "<<n2<<"\n";
}
void cube() // déclaration de la fonction
{
int n, n3; // variables locales à cube
cout<<"ENTRER UN NOMBRE : "; cin>>n;
n3 = n*n*n;
cout<<"VOICI SON CUBE : "<<n3<<"\n";
}
void main() // programme principal
{
char choix; // variable locale à main()
cout<<"CALCUL DU CARRE TAPER 2\n";
cout<<"CALCUL DU CUBE TAPER 3\n";
cout<<"\nVOTRE CHOIX : "; cin>>choix;
```

```
switch (choix)
{
case '2':carre();
break;
case '3':cube();
break;
}
getch();
}
```

Exercice 3:

quells résultats fournira ce programme :

```
#include <iostream>
#include<conio.h>
int n=10, q=2 ;
main()
{
int fct (int) ;
void f (void) ;
int n=0, p=5 ;
n = fct(p) ;
cout << "A : dans main, n = " << n << " p = " << p
<< " q = " << q << "\n" ;
f() ;
}
int fct (int p)
{
int q ;
q = 2 * p + n ;
cout << "B : dans fct, n = " << n << " p = " << p
<< " q = " << q << "\n" ;
return q ;
}
void f (void)
{
int p = q * n ;
cout << "C : dans f, n = " << n << " p = " << p
<< " q = " << q << "\n" ;
}
```

Travaux pratiques (TP°5)

Exercice 1 :

Comment concevoir le type classe chose de façon que ce petit programme :

```
main()
{ chose x ;
cout << "bonjour\n" ;
}
```

fournisse les résultats suivants :

création objet de type chose

bonjour

destruction objet de type chose

Que fournira alors l'exécution de ce programme (utilisant le même type chose) :

```
main()
{ chose * adc = new chose
}
```

Exercice 2 :

Créer une classe point ne contenant qu'un constructeur sans arguments, un destructeur et un membre donnée privé représentant un numéro de point (le premier créé portera le numéro 1, le suivant le numéro 2...). Le constructeur affichera le numéro du point créé et le destructeur affichera le numéro du point détruit. Écrire un petit programme d'utilisation créant dynamiquement un tableau de 4 points et le détruisant.

ANNEXE

1. Quelques fichiers d'entêtes du C++ :

#include<iostream> : il est utilisé comme flux d'entrée et de sortie à l'aide de cin et cout.

#include<string.h> : Il est utilisé pour exécuter diverses fonctionnalités liées à la manipulation de strings comme strlen() , strcmp() , strcpy() , size(), etc.

#include<math.h> : Il est utilisé pour effectuer des opérations mathématiques comme sqrt() , log2() , pow() , etc.

#include<iomanip.h> : Il est utilisé pour accéder aux fonctions set () et setprecision() pour limiter les décimales dans les variables.

#include<signal.h> : Il est utilisé pour exécuter des fonctions de gestion de signal comme signal() et raise() .

#include<stdarg.h> : Il est utilisé pour exécuter des fonctions d'argument standard comme va_start() et va_arg() . Il est également utilisé pour indiquer le début de la liste d'arguments de longueur variable et pour extraire les arguments de la liste d'arguments de longueur variable dans le programme respectivement.

#include<errno.h> : Il est utilisé pour effectuer des opérations de gestion des erreurs comme errno() , strerror() , perror() , etc.

#include<fstream.h> : Il est utilisé pour contrôler les données à lire à partir d'un fichier en tant qu'entrée et les données à écrire dans le fichier en tant que sortie.

#include<time.h> : Il est utilisé pour exécuter des fonctions liées à date() et time() comme setdate() et getdate() . Il est également utilisé pour modifier la date système et obtenir le temps CPU respectivement.

2. Types de variables :

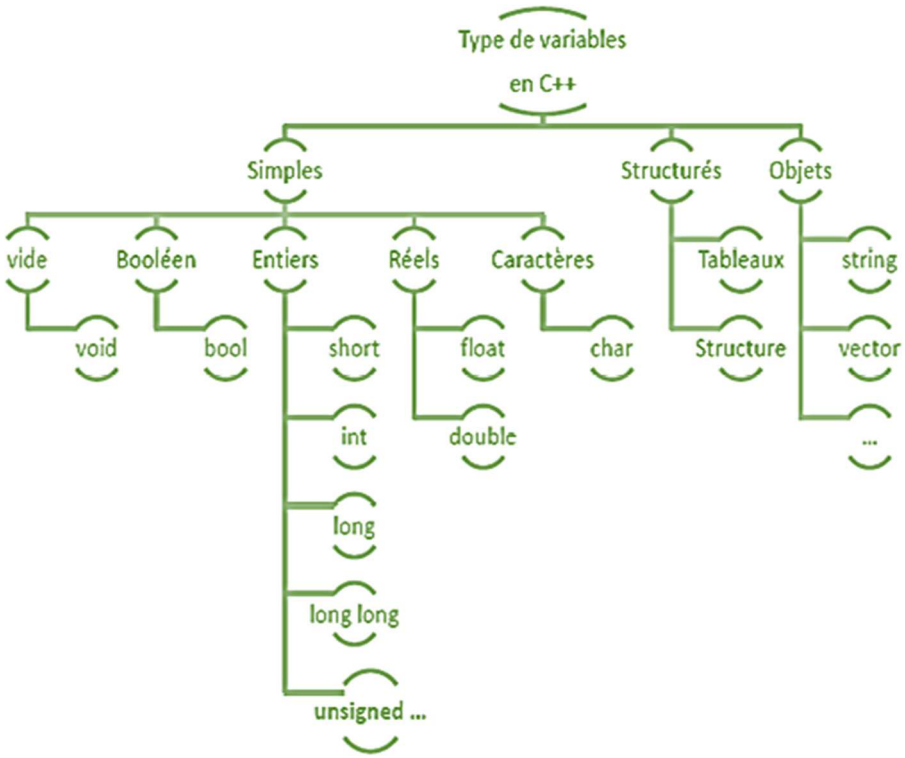


Figure 5. Diffèrent type de variable

3. Les mots- clés : (également appelés mots réservés)

Ont une signification particulière pour le compilateur C++ et sont toujours écrits ou tapés en minuscules. Les mots clés sont des mots que le langage utilise dans un but particulier, tels que void , int , public , etc. Il ne peut pas être utilisé pour un nom de variable ou un nom de fonction. Vous trouverez ci-dessous le array de l'ensemble complet de mots-clés C++.

Mot-clé C++			
asm	double	**Nouveau**	changer
auto	autre	**opérateur**	**modèle**
Pause	énumération	**privé**	**cette**
Cas	externe	**protégé**	**jeter**
attraper	flotteur	**Publique**	**essayer**
carboniser	pour	S'inscrire	typedef
classer	**ami**	revenir	syndicat
constante	aller à	court	non signé
Continuez	si	signé	**virtuel**
défaut	**en ligne**	taille de	annuler
effacer	entier	statique	volatil
fais	long	structure	tandis que

Table 8. Quelques mots réservés au C++

Remarque : Les mots-clés introuvables dans ANSI C sont affichés ici en gras.

- **asm** : Pour déclarer qu'un bloc de code doit être passé à l'assembleur.
- **auto** : un spécificateur de classe de stockage utilisé pour définir des objets dans un bloc.
- **break** : Termine une instruction switch ou une boucle.
- **case** : utilisé spécifiquement dans une instruction switch pour spécifier une correspondance pour l'expression de l'instruction.
- **catch** : spécifie les actions entreprises lorsqu'une exception se produit.
- **char** : Type de données fondamental qui définit les objets caractère.
- **class** : pour déclarer un type défini par l'utilisateur qui encapsule des membres de données et des opérations ou des fonctions membres.
- **const** : Pour définir des objets dont la valeur ne changera pas tout au long de la durée de vie de l'exécution du programme.
- **continue** :- Transfère le contrôle au début d'une boucle.
- **default** :- Gère les valeurs d'expression dans une instruction switch qui ne sont pas gérées par la casse.
- **delete** : Opérateur de désallocation de mémoire.
- **do** : indique le début d'une instruction do-while dans laquelle la sous-instruction est exécutée de manière répétée jusqu'à ce que la valeur de l'expression soit logical-false.
- **double :** type de données fondamental utilisé pour définir un nombre à virgule flottante.

- **else :** utilisé spécifiquement dans une instruction if-else.
- **enum :** pour déclarer un type de données d'énumération défini par l'utilisateur.
- **extern** : un identifiant spécifié comme extern a un lien externe avec le bloc.
- **float** :- Type de données fondamental utilisé pour définir un nombre à virgule flottante.
- **for** : Indique le début d'une instruction pour réaliser un contrôle répétitif.
- **ami :** une classe ou une opération dont l'implémentation peut accéder aux données privées membres d'une classe.
- **goto** : Transfère le contrôle à une étiquette spécifiée.
- **if** : indique le début d'une instruction if pour obtenir un contrôle sélectif.
- **inline :** un spécificateur de fonction qui indique au compilateur que la substitution en ligne du corps de la fonction doit être préférée à l'implémentation habituelle de l'appel de fonction.
- **int :** type de données fondamental utilisé pour définir des objets entiers.
- **long :** un modificateur de type de données qui définit un int 32 bits ou un double étendu.
- **new** : Opérateur d'allocation de mémoire.
- **operator :** surcharge un opérateur c^{++} avec une nouvelle déclaration.
- **private** : Déclare les membres de la classe qui ne sont pas visibles en dehors de la classe.
- **protected :** Déclare les membres de la classe qui sont privés, sauf pour les classes dérivées
- **public :** déclare les membres de la classe qui sont visibles en dehors de la classe.
- **registre :** un spécificateur de classe de stockage qui est un spécificateur automatique, mais qui indique également au compilateur qu'un objet sera fréquemment utilisé et doit donc être conservé dans un registre.
- **return** : renvoie un objet à l'appelant d'une fonction.
- **short :** un modificateur de type de données qui définit un nombre entier de 16 bits.
- **signé :** un modificateur de type de données qui indique que le signe d'un objet doit être stocké dans le bit de poids fort.
- **sizeof :** renvoie la taille d'un objet en octets.
- **statique :** la durée de vie d'un objet statique défini existe tout au long de la durée de vie de l'exécution du programme.
- **struct :** pour déclarer de nouveaux types qui encapsulent à la fois des données et des fonctions membres.

- **switch** : Ce mot-clé utilisé dans l'« instruction Switch ».
- **template** : type paramétré ou générique.
- **this** : Un pointeur de classe pointe vers un objet ou une instance de la classe.
- **lancer :** génère une exception.
- **try** : Indique le début d'un bloc de gestionnaires d'exceptions.
- **typedef** : Synonyme d'un autre type intégral ou défini par l'utilisateur.
- **union :** Semblable à une structure, struct, en ce qu'elle peut contenir différents types de données, mais une union ne peut contenir qu'un seul de ses membres à un moment donné.
- **non signé :** un modificateur de type de données qui indique que le bit de poids fort doit être utilisé pour un objet.
- **virtual** : un spécificateur de fonction qui déclare une fonction membre d'une classe qui sera redéfinie par une classe dérivée.
- **void :** Absent d'une liste de paramètres de type ou de fonction.
- **volatile** : définit un objet dont la valeur peut varier d'une manière indétectable par le compilateur.
- **while** : début d'une instruction while et fin d'une instruction do-while

4. Les fonctions mathématiques <math.h>

4.1. Fonctions trigonométriques et hyperboliques :

Fonction	Description
acos	**Arc cosinus**
asin	**Arc sinus**
atan	**Arc tangente**
cos	**Cosinus**
cosh	**Cosinus hyperbolique**
sin	**sinus**
sinh	**sinus hyperbolique**
tan	**tangente**
tanh	**tangente hyperbolique**

Table 9. Les Fonctions trigonométriques et hyperboliques

4.2. Fonctions exponentielles et logarithmiques :

exp	exponentielle
frexp	etant donne x, trouver n et p tels que $x=n*2^p$
ldexp	multiplie un nombre par une puissance entiere de 2
log	logarithme neperien
log10	logarithme en base 10
modf	calcul la partie entiere et la partie decimale d'un flottant

Table 10. Les Fonctions exponentielles et logarithmiques

4.3. Fonctions diverses :

Ceil	Arrondi à l'entier le plus proche par la valeur supérieur
Floor	Arrondi à l'entier le plus proche par la valeur inférieur
Fabs	Valeur absolue
Fmod	Reste de la division
pow	Puissance x^y
sqrt	Racine carrée

Table 11. Les Fonctions diverses

Bibliographie

[1] Delannoy Claude, Exercices en langage, C++ , Ed3. Eyrolles 2007

[2] Bjarne Stroustrup, Marie-Cécile Baland, Emmanuelle Burr, Christine Eberhardt, Programmation : Principes et pratique avec C++, Edition Pearson 2012.

[3] Jean-Cédric Chappelier, Florian Seydoux, C++ par la pratique. Recueil d'exercices corrigés et aide-mémoire, PPUR Édition : 3e édition 2012.

[4] Jean-Michel Léry, Frédéric Jacquenot, Algorithmique, applications aux langages C, C++ en Java Edition Pearson, 2013.

[5] Frédéric Drouillon, Du C au C++ - De la programmation procédurale à l'objet, Eni; Édition :2e édition 2014.

[6] Claude Delannoy, Programmer en langage C++, Edition Eyrolles 2000.

[7] Kris Jamsa, Lars Klander, C++ La bible du Programmeur, Edition Eyrolles 2000.

[8] Bjarne Stroustrup, Le Langage C++, Édition Addison-Wesley 2000.

[9] https://public.iutenligne.net/informatique/langages/maillefert/LangageC /cours/cours.pdf

[10] https://openclassrooms.com/courses/programmez-avec-le-langage-c/les-tableaux-5

[11] http://www.fresnel.fr/perso/stout/langage_C/Chap_8_Tableaux_de_char.pdf

[12] A. Azough, Cours du Langage C++, Université Sidi Mohamed Ben Abdellah Faculté des sciences, Maroc, 2017

[13] Bartjan van Tent, COURS entrées, sorties, fichiers, en C++, Université Paris-Sud Orsay, L3 et Magistère 1ère année de Physique Fondamentale, 2015

[14] S. Laporte , Cours C++ : le fichiers, BTS IG 1 Lycée Louise Michel, 2015-2016

[15] T. Redarce, N. Ducros, O. Bernard, Langage C/C++, De la syntaxe à la programmation orientée objet, T. Grenier, November 21, 2017

[16] F. MALIKI , Cours Algorithmique 2 : Techniques de programmation orientée objet, EPST Tlemcen 2014

[17] Belaid, Programmation en C++,COURS +TD+TP , Edition pages bleues 2016.

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

[19] https://www.tutorialspoint.com/cplusplus/index.html