Chapitre 1
Présentation du langage C++

1.1 Introduction

Le langage C est un langage évolué et structuré, assez proche du langage machine destiné à des applications de contrôle de processus (gestion d'entrées/sorties, applications temps réel, ...). Né dans les laboratoires Bell en 1972. Les compilateurs C possèdent les taux d'expansion les plus faibles de tous les langages évolués (rapport entre la quantité de codes machine générée par le compilateur et la quantité de codes machine générée par l'assembleur et ce pour une même application).

1.2 Historique de C++

Très tôt, les concepts de la programmation orientée objet (en abrégé P.O.O.) ont donné naissance à de nouveaux langages dits « orientés objets » tels que Smalltalk, Simula, Eiffel ou, plus récemment, Java. Le langage C++, quant à lui, a été conçu suivant une démarche hybride. En effet, Bjarne Stroustrup, son créateur, a cherché à adjoindre à un langage structuré existant (le C), un certain nombre de spécificités lui permettant d'appliquer les concepts de P.O.O. Dans une certaine mesure, il a permis à des programmeurs C d'effectuer une transition en douceur de la programmation structurée vers la P.O.O. De sa conception jusqu'à sa normalisation, le langage C++ a quelque peu évolué. Initialement, un certain nombre de publications de AT&T ont servi de référence du langage. Les dernières en date sont : la version 2.0 en 1989, les versions 2.1 et 3 en 1991. C'est cette dernière qui a servi de base au travail du comité ANSI qui, sans la remettre en cause, l'a enrichie de quelques extensions et surtout de composants standard originaux se présentant sous forme de fonctions et de classes génériques qu'on désigne souvent par le sigle S.T.L (Standard Template Library). La norme définitive de C++ a été publiée par l'ANSI en juillet 1998. 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

stdio.h : bibliothèque d'entrées/sorties standard

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

Comment traduire

- Compilation (traduire avant l'exécution)
- Interprétation (traduire pendant l'exécution)

Les compilateurs C sont remplacés petit à petit par des compilateurs C++.

Un programme écrit en C est en principe compris par un compilateur C++.

1.3 Etapes permettant l'edition, la Mise Au Point, l'execution d'un programme

*1- **Edition du programme source***, à l'aide d'un éditeur (traitement de textes). Le nom du fichier contient l'extension .CPP, exemple: EXI_1.CPP (menu « edit »).

*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 listable, appelé fichier objet: EXI_1.OBJ (commande « compile »).

*3- **Editions de liens***: Le code machine des fonctions-bibliothèque est chargé, création d'un fichier binaire, non listable, appelé fichier executable: EXI_1.EXE (commande « build all »).

*4- **Exécution du programme*** (commande « 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 ...

Exercice I-1: Editer (EXI_1.CPP), compiler et exécuter le programme suivant:

```
#include <stdio.h>          /* bibliotheque d'entrees-sorties standard */
#include <conio.h>
void main()
{
puts("BONJOUR");/* utilisation d'une fonction-bibliotheque */
puts("Pour continuer frapper une touche...");
getch();  /* Attente d'une saisie clavier */}
```

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.

Dans ce programme, on introduit 3 nouveaux concepts :

- La notion de déclaration de variables : les variables sont les données que manipulera le programme lors de son exécution. Ces variables sont rangées dans la mémoire vive de l'ordinateur. Elle doivent être déclarées au début du programme.

- La notion d'affectation, symbolisée par le signe =.

- La notion d'opération.

Chapitre 2
Syntaxe élémentaire en langage C++

2.1 Introduction

Le C++ est un langage procédural, du même type que le Pascal par exemple. Cela signifie que les instructions sont exécutées linéairement et regroupées en blocs : les fonctions et les procédures (les procédures n'existent pas en C++, ce sont des fonctions qui ne retournent pas de valeur).

2.2 Les commentaires en C++

Les commentaires sont nécessaires et très simples à faire. Tout programme doit être commenté. Attention cependant, trop de commentaires tue le commentaire, parce que les choses importantes sont noyées dans les banalités.
Il existe deux types de commentaires en C++ : les commentaires de type C et les commentaires de fin de ligne (qui ne sont disponibles qu'en C++).
Les commentaires C commencent avec la séquence barre oblique - étoile. Les commentaires se terminent avec la séquence inverse : une étoile suivie d'une barre oblique.

Exemple 2-1. Commentaire C

```
/* Ceci est un commentaire C */
```

Ces commentaires peuvent s'étendre sur plusieurs lignes.

Exemple 2-2. Commentaire C++

```
action quelconque // Ceci est un commentaire C++
action suivante
```

Dans la suite, les programmes d'exemples utiliseront les deux formes de commentaires pour des raisons de clarté. Si les programmes doivent être compilés avec un compilateur C, il faut supprimer les commentaires spécifiques C++.

2.3 Les types prédéfinis du C/C++

Il y a plusieurs types prédéfinis. Ce sont :
• le type vide : void. Ce type est utilisé pour spécifier le fait qu'il n'y a pas de type. Ceci a une utilité pour faire des procédures (fonctions ne renvoyant rien) et les pointeurs sur des données non typées;

• les booléens : bool, qui peuvent prendre les valeurs true et false (en C++ uniquement, ils n'existent pas en C) ;
• les caractères : char ;
• les caractères longs : wchar_t (en C++ seulement, ils n'existent pas en C) ;
• les entiers : int ;
• les réels : float ;
• les réels en double précision : double ;
• les tableaux à une dimension, dont les indices sont spécifiés par des crochets ('[' et ']'). Pour les
tableaux de dimension supérieure ou égale à 2, on utilisera des tableaux de tableaux ;

• les structures, unions et énumérations.

2.4 Les différents types de variables

1- Les entiers

Le langage C distingue plusieurs types d'entiers:

TYPE	DESCRIPTION	TAILLE MEMOIRE
int	entier standard signé	4 octets: $-2^{31} \leq n \leq 2^{31}-1$
unsigned int	entier positif	4 octets: $0 \leq n \leq 2^{32}$
short	entier court signé	2 octets: $-2^{15} \leq n \leq 2^{15}-1$
unsigned short	entier court non signé	2 octets: $0 \leq\leq n \leq 2^{16}$
char	caractère signé	1 octet : $-2^{7} \leq\leq n \leq 2^{7}-1$
unsigned char	caractère non signé	1 octet : $0 \leq n \leq 2^{8}$

Numération: En décimal les nombres s'écrivent tels que, précédés de 0x en hexadécimal.

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

Remarque: En langage C, le type **char** est un cas particulier du type entier:

un caractère est un entier de 8 bits

Exemples:

Les caractères alphanumériques s'écrivent entre ‘ ‘

Le **caractère** 'b' a pour valeur 98 (son code ASCII).

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 charriot	0x0d	CR
'\f'	saut de page	0x0c	FF

'\\'	backslash	0x5c	\
'\"	cote	0x2c	'
'\"'	guillemets	0x22	

Modifier ainsi le programme et le tester :

```
#include <stdio.h>              /* bibliotheque d'entrees-sorties standard */
#include <conio.h>
void main()
{

int a, b, calcul ;  /* déclaration de 3 variables */
char u, v;
puts("BONJOUR");   /* utilisation d'une fonction-bibliotheque */
a = 10 ; /* affectation* /
b = 50 ;  /* affectation */
u = 65 ;
v = 'A' ;
calcul = (a + b)*2 ;  /* affectation et opérateurs   */
printf(« Voici le resultat : %d\n », calcul) ;
printf(« 1er affichage de u : %d\n »,u) ;
printf(« 2ème affichage de v : %c\n »,u) ;
 printf(« 1er affichage de u: %d\n »,v) ;
printf(« 2éme  affichage de v: %c\n »,v) ;
puts("Pour continuer frapper une touche...");
getch();  /* Attente d'une saisie clavier */
}
```

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 dans la zône des 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 ...

Formats de sortie pour les entiers:

%d affichage en décimal (entiers de type int),
%x affichage en hexadécimal (entiers de type int),
%u affichage en décimal (entiers de type unsigned int),
D'autres formats existent, consulter une documentation constructeur.

Exercice I-5:

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

Exercice I-6:

Que va-t-il se produire, à l'affichage, lors de l'exécution du programme suivant ?

AUTRES FONCTIONS DE SORTIES

Affichage d'un caractère: La fonction **putchar** permet d'afficher un caractère:
c étant une variable de type **char**, l'écriture **putchar(c);** est équivalente à **printf("%c\n",c);**

Affichage d'un texte: La fonction **puts** permet d'afficher un texte:
l'écriture **puts("bonjour");** est équivalente à **printf("bonjour\n");**

Il vaut mieux utiliser puts et putchar si cela est possible, ces fonctions, non formatées, sont d'exécution plus rapide, et nécessitent moins de place en mémoire lors de leur chargement.

LES OPERATEURS

Opérateurs arithmétiques sur les réels: + - * / avec la hiérarchie habituelle.
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.
Opérateurs logiques sur les entiers:
& ET | OU ^ OU EXCLUSIF ~ COMPLEMENT A UN « DECALAGE A GAUCHE
» DECALAGE A DROITE.

```
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 */
```

L'opérateur **sizeof(type)** renvoie le nombre d'octets réservés en mémoire pour chaque type d'objet.

```
Exemple: n = sizeof(char);   /* n vaut 1 */
```

INCREMENTATION - DECREMENTATION

Le langage C autorise des écritures simplifiées pour l'incrémentation et la décrémentation de

variables:
i = i+1; est équivalent à i++;
i = i-1; est équivalent à i--;

OPERATEURS COMBINES

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;

LES DECLARATIONS DE CONSTANTES

Le langage C autorise 2 méthodes pour définir des constantes.

1ere méthode: **déclaration** d'une variable, dont la valeur sera constante pour tout le programme:

Exemple:
```
void main()
{
const float PI = 3.14159;
float perimetre,rayon = 8.7;
perimetre = 2*rayon*PI;
....
}
```

Dans ce cas, le compilateur réserve de la place en mémoire (ici 4 octets), pour la variable pi, mais dont on ne peut changer la valeur.

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

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

Le compilateur ne réserve pas de place en mémoire. Les constantes déclarées par #define s'écrivent traditionnellement en majuscules, mais ce n'est pas une obligation.

LES CONVERSIONS DE TYPES

Le langage C permet d'effectuer des opérations de conversion de type: On utilise pour cela l'opérateur de "cast" ().

Chapitre 3
Structures conditionnelles et Boucles

Dans ce chapitre, nous allons étudier plusieurs instructions qui permettent de faire varier le comportement des programmes en fonction de certaines conditions.

3.1 Tests: instructions conditionnelles (1)

Les instructions conditionnelles servent à n'exécuter une instruction ou une séquence d'instructions que si une condition est vérifiée

On utilisera la forme suivante:

```
if  (condition ){
instruction ou suite d'instructions1}
else{
        instruction ou suite d'instructions2}
```

- la condition ne peut être que vraie ou fausse
- si la condition est vraie, se sont les instructions1 qui seront exécutées
- si la condition est fausse, se sont les instructions2 qui seront exécutées
- la condition peut être une condition simple ou une condition composée de plusieurs conditions

3.2 Tests: instructions conditionnelles (2)

La partie else n'est pas obligatoire, quand elle n'existe pas et que la condition est fausse, aucun traitement n'est réalisé

- On utilisera dans ce cas la forme simplifiée suivante:

```
if (condition){
        instruction ou suite d'instructions1
}
```

Exemple:

SI (score>meilleur_score) **ALORS**
meilleur_score ← score;

En C++ :

```
IF (score>meilleur_score)
    meilleur_score = score;
```

3.3 Instructions itératives: les boucles

Les boucles servent à répéter l'exécution d'un groupe d'instructions un certain nombre de fois
On distingue trois sortes de boucles en langages de programmation :
Les boucles while (tant que) : on y répète des instructions tant qu'une certaine condition est réalisée
Les boucles Boucle do-while (jusqu'à) : on y répète des instructions jusqu'à ce qu'une certaine condition soit réalisée
Les boucles for (pour) ou avec compteur : on y répète des instructions en faisant évoluer un compteur (variable particulière) entre une valeur initiale et une valeur finale.

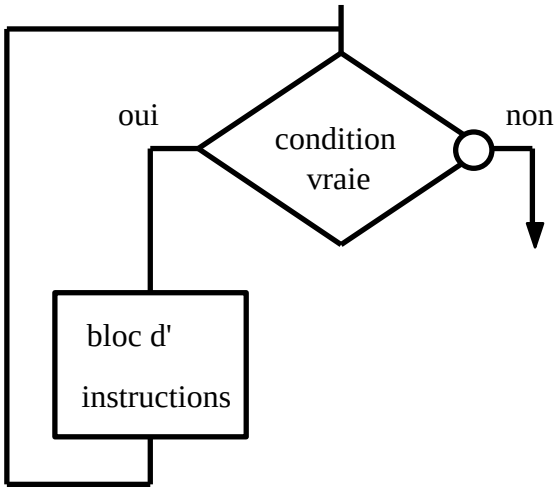
3.3.1 Les boucles while

Il s'agit de l'instruction:

```
while    (condition)
{
        instructions

}
```

Organigramme:



la condition (dite condition de contrôle de la boucle) est évaluée avant chaque itération
si la condition est vraie, on exécute instructions (corps de la boucle), puis, on retourne tester la condition. Si elle est encore vraie, on répète l'exécution, …
si la condition est fausse, on sort de la boucle et on exécute l'instruction qui est après Fin de boucle

Remarques

Le nombre d'itérations dans une boucle while n'est pas connu au moment d'entrée dans la boucle. Il dépend de l'évolution de la valeur de condition
Une des instructions du corps de la boucle doit absolument changer la valeur de condition de vrai à faux (après un certain nombre d'itérations), sinon le programme tourne indéfiniment , Attention aux boucles infinies.

Exemple

Afficher les nombres entiers dont le carré est inférieur à 100.

```
nombre ← 1;
TANT QUE (nombre*nombre<100) FAIRE
    afficher(nombre);
    nombre ← nombre + 1;
FIN TANT QUE
```

En C++

```
nombre = 1;
while (nombre*nombre<100)
 {
cout << nombre << endl;
 nombre=nombre+1;
   }
```

3.3.2 Les boucles for for

Syntaxe en C:

for(initialisation ; condition de continuité ; modification)

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

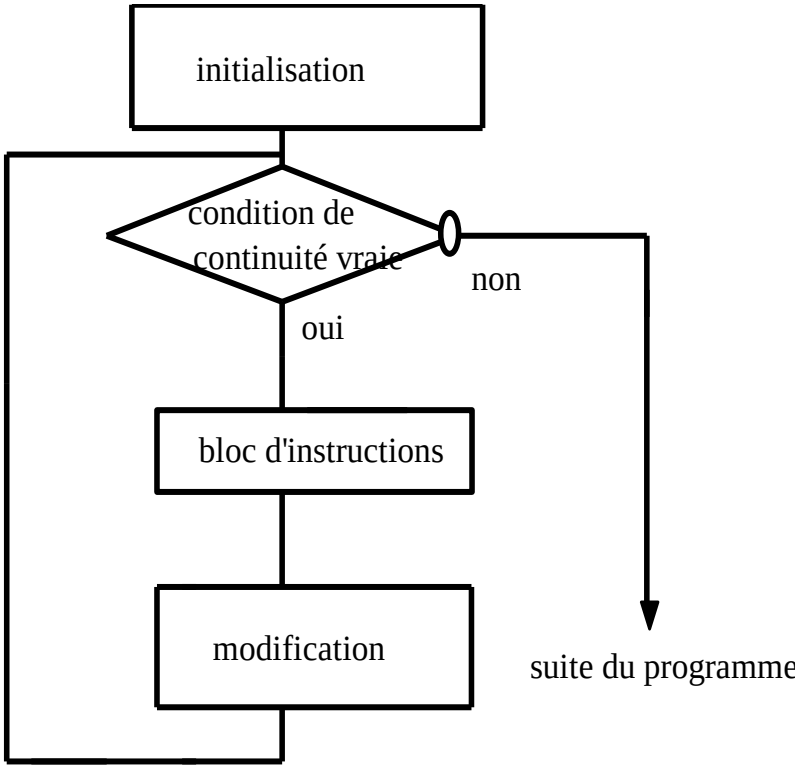
Remarques:

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 ;

Organigramme:



Remarque : le nombre d'itérations dans une boucle for est connu avant le début de la boucle

Compteur est une variable de type entier (ou caractère). Elle doit être déclarée

Pas est un entier qui peut être positif ou négatif. **Pas** peut ne pas être mentionné, car par défaut sa valeur est égal à 1. Dans ce cas, le nombre d'itérations est égal à finale - initiale+ 1

Initiale et finale peuvent être des valeurs, des variables définies,

3.3.2.1 Déroulement des boucles for

1) La valeur initiale est affectée à la variable compteur

    1) On compare la valeur du compteur et la valeur de finale :

b) Si la valeur du compteur est > à la valeur finale dans le cas d'un pas positif (ou si compteur est < à finale pour un pas négatif), on sort de la boucle et on continue avec l'instruction qui suit fin boucle for

c) Si compteur est <= à finale dans le cas d'un pas positif (ou si compteur est >= à finale pour un pas négatif), instructions seront exécutées

    i. Ensuite, la valeur de compteur est incrémentée de la valeur du pas si pas est positif (ou décrémenté si pas est négatif)

ii. On recommence l'étape 2 : La comparaison entre compteur et finale est de nouveau effectuée, et ainsi de suite …

Exemple

POUR i←1 à 10 FAIRE

afficher(i);

afficher(i*i);

FIN POUR

En C++

```
for (i=1; i<=10; i++)
{   cout << i << endl;
   cout << i*i << endl;
}
```

Remarque

Il faut éviter de modifier la valeur du compteur (et de finale) à l'intérieur de la boucle. En effet, une telle action :

- perturbe le nombre d'itérations prévu par la boucle for
- rend difficile la lecture de l'algorithme
- présente le risque d'aboutir à une boucle infinie

Exemple :

```
for (i=1; i<=10; i++)
        {       cout << i << endl;
                cin << i; }
```

3.3.3 Les boucles do/while

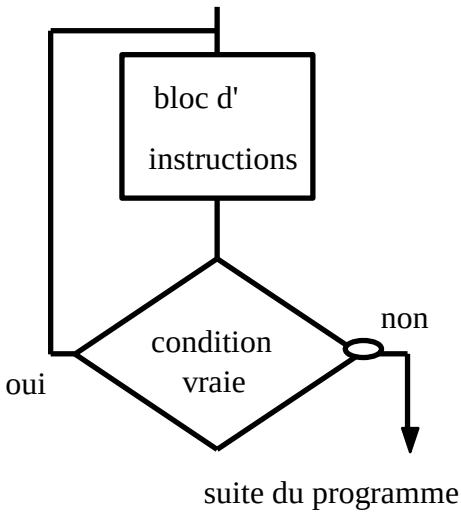
Syntaxe en C:

```
    do{
       instructions     }
 while (condition);
```

Condition est évaluée après chaque itération

les instructions entre *do* et *while* sont exécutées au moins une fois et leur exécution est répétée jusqu'à ce que condition soit vrai (tant qu'elle est fausse)

Organigramme:



Exemple

Un algorithme qui détermine le premier nombre entier N tel que la somme de 1 à N dépasse strictement 100 **(version avec répéter jusqu'à)**

Debut
som ← 0
i ← 0
Répéter
i ← i+1
som ← som+i
Jusqu'à (som > 100)
Fin

En C++

```
Som=0;
i=0;
do {
i=i+1;
Som=som+i;}
While(som>0)
```

3.4 Choix d'un type de boucle

Si on peut déterminer le nombre d'itérations avant l'exécution de la boucle, il est plus naturel d'utiliser *la boucle for*

S'il n'est pas possible de connaître le nombre d'itérations avant l'exécution de la boucle, on fera appel à l'une des *boucles while ou while/do*

Pour le choix entre *while et* do/while *:*

Si on doit tester la condition de contrôle avant de commencer les instructions de la boucle, on utilisera *while*

Si la valeur de la condition de contrôle dépend d'une première exécution des instructions de la boucle, on utilisera do/while,

Chapitre 4
Entrées/sorties

4.1 Introduction

Les entrées/sorties (Input/Output ou I/O) permettent à un programme de communiquer avec certains périphériques.
- Les entrées/sorties permettent par exemple :
  - de lire et d'écrire sur la console (lecture au clavier et affichage à l'écran en mode 'caractères')
  - d'accéder aux fichiers et répertoires des disques
  - de communiquer avec d'autres applications (localement ou au travers du réseau)

4.2 Flux (Stream)

La notion de flux (Stream) caractérise un chemin (une voie) de communication entre une source d'information et sa destination.
L'accès aux informations s'effectue de manière séquentielle.
Les librairies Java distinguent deux genres de flux :
• Les flux binaires (Byte Stream) qui peuvent représenter des données quelconques (nombres, données structurées, programmes, sons, images, …)
• Les flux de caractères (Character Stream) qui représentent des textes (chaînes de caractères) au format Unicode.

La communication avec un flux comprend trois phases : •
L'ouverture du flux, en entrée (pour la lecture) ou en sortie (pour l'écriture)
La répétition de la lecture ou de l'écriture des données (généralement des bytes ou des caractères)
La fermeture du flux

4.3 LES CHAINES DE CARACTERES

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:

éclaration: **char nom[dim];** ou bien **char *nom;**

nom = (char*)malloc(dim);

Exemple: **char texte[dim];** ou bien **char *texte;**

texte = (char*)malloc(10);

Le compilateur réserve (dim-1) places en mémoire pour la chaîne de caractères: En effet, il ajoute toujours le caractère **NUL ('\0')** à la fin de la chaîne en mémoire.

Écriture sur la sortie standard

Écriture sur la sortie standard Fonction printf et opérateur <<

Exemple 1

```
#include<iostream> // indispensable pour utiliser  cout

 int main() {

char texte[10] = « BONJOUR »;

cout << texte;

 return 0; }
```

cout est un flot de sortie prédéfini associé à la sortie standard (stdout du C).

- << est un opérateur dont l'opérande de gauche (cout) est un flot et l'opérande de droite, une expression de type quelconque.

L'instruction "cout << …" précédente peut être interprétée comme ceci : le flot cout reçoit la valeur "bonjour".

Exemple 2

```
#include <iostream> // indispensable pour utiliser cout

int main() {

int n = 25;

char c = 'a';

 char *ch = "bonjour";

double x = 12.3456789;

cout << "valeur de n: "<< n<<"\n ";

cout << "valeur de n: "<< c<<"\n ";

cout << "valeur de n: "<< ch<<"\n ";

cout << "valeur de n: "<< x<<"\n ";

 return 0; }
```

L'exécution devrait donner :

valeur de n : 25, valeur de c : a, chaîne ch : bonjour , valeur de x : 12.3456789

Lecture sur l'entrée standard

Fonction scanf et opérateur >>

Lecture en C :

```
#include
…
scanf ( Format, Liste des adresses des variables);
```

Lecture en C++ :

```
#include <iostream.h>
…
cin >> Var1 >> Var2 >> … >> VarN;
```

cin est le flot d'entrée connectée à l'entrée standard, le clavier. Il correspond au fichier prédéfini stdin du langage C.

Exemples

```
#include <iostream.h>
int main ()
{ int n;
float x;
char t[80+1];
do {
 cout << "donner un entier, une chaîne et un flottant : ";
 cin >> n >> t >> x;
cout << "merci pour " << n << ", " << t << " et " << x << "\n";
} while (n);
 return 0;}
```

Exécution:

donner un entier, une chaîne et un flottant : 15 bonjour 8.25

merci pour 15, bonjour et 8.25

donner un entier, une chaîne et un flottant : 15 bonjour 8.25

merci pour 15, bonjour et 8.25

donner un entier, une chaîne et un flottant : 0 bye 0 merci pour 0, bye et 0

Chapitre 5
Pointeurs et Tableaux

5.1 Introduction

L'étude des pointeurs montre l'adaptation du langage C++ à la conduite de processus. On verra dans ce chapitre et les suivants la puissance de cette notion par ailleurs de concept simple pour un informaticien industriel.

5.2 L'OPERATEUR ADRESSE &

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

Exemple: LES POINTEURS

```
char x = 'A'; // création de la variable char en pile
char* p = &x; // création de la variable pointeur sur char en pile et affectation de l'adresse de la variable en pile
```

Définition: Un pointeur est une adresse mémoire. On dit que le pointeur pointe sur cette adresse.

DECLARATION DES POINTEURS

Une variable de type pointeur se déclare à l'aide de l'objet pointé précédé du symbole *** (opérateur d'indirection).**

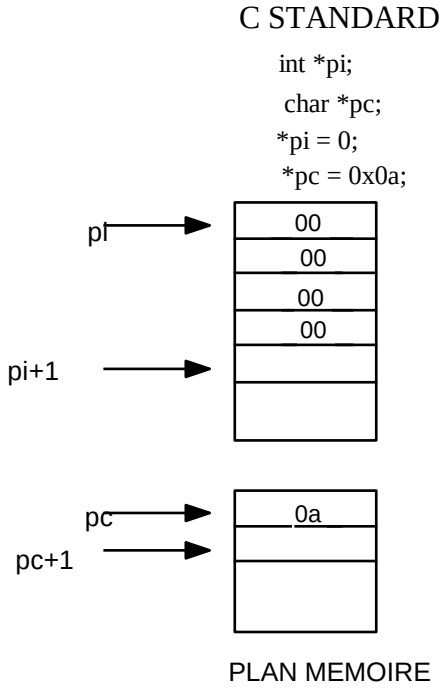
Exemple:
```
char *pc;    pc est un pointeur pointant sur un objet de type char
int *pi;     pi est un pointeur pointant sur un objet de type int
float *pr;   pr est un pointeur pointant sur un objet de type float
```

L'opérateur * désigne en fait le contenu de l'adresse.

Exemple:
```
char *pc;
*pc = 34 ;
printf("CONTENU DE LA CASE MEMOIRE: %c\n",*pc);
printf("VALEUR DE L'ADRESSE EN HEXADECIMAL: %p\n",pc);
```

5.3 ARITHMETIQUE DES POINTEURS

On peut essentiellemet **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  */

pr++;             /* incrémente la valeur du pointeur r (de 4 cases mémoire) */

printf("L'ADRESSE r VAUT: %p\n",pr);          /* affichage de la valeur de l'adresse r */

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

pc--;             /* décrémente la valeur du pointeur c (d'une case mémoire) */
```

5.4 ALLOCATION DYNAMIQUE

Lorsque l'on déclare une variable char, int, float .... un nombre de cases mémoire bien défini est **réservé** pour cette variable. Il n'en est pas de même avec les pointeurs.

Exemple:

```
char *pc;

*pc = 'a';        /* le code ASCII de a est rangé dans la case mémoire pointée par c */

*(pc+1) = 'b';  /* le code ASCII de b  est rangé une case mémoire plus loin */

*(pc+2) = 'c';  /* le code ASCII de c est rangé une case mémoire plus loin */

*(pc+3) = 'd';  /* le code ASCII de d est rangé une case mémoire plus loin */
```

Dans cet exemple, le compilateur a attribué une valeur au pointeur c, les adresses suivantes sont donc bien définies; mais le compilateur n'a pas **réservé** ces places: il pourra très bien les attribuer un peu plus tard à d'autres variables. Le contenu des cases mémoires c c+1 c+2 c+3 sera donc perdu.

Ceci peut provoquer un blocage du système sous WINDOWS.

Il existe en langage C, des fonctions permettant **d'allouer de la place en mémoire** à un pointeur.

Il faut absolument les utiliser dès que l'on travaille avec les pointeurs.

Exemple: la fonction **malloc**

```
char *pc;

int *pi,*pj,*pk;

float *pr;

pc = (char*)malloc(10);        /* on réserve 10 cases mémoire, soit la place pour 10 caractères
*/

pi = (int*)malloc(16);         /* on réserve 16 cases mémoire, soit la place pour 4 entiers  */

 pr = (float*)malloc(24);      /* on réserve 24 places en mémoire, soit la place pour 6 réels */

pj = (int*)malloc(sizeof(int));          /* on réserve la taille d'un entier en mémoire */

pk = (int*)malloc(3*sizeof(int));      /* on réserve la place en mémoire pour 3 entiers */
```

Libération de la mémoire: la fonction **free**

```
free(pi);                        /* on libère la place précédemment réservée pour i */
```

free(pr); /* on libère la place précédemment réservée pour r */

Exercice V_2:

adr1 et adr2 sont des pointeurs pointant sur des réels. Le contenu de adr1 vaut

-45,78; le contenu de adr2 vaut 678,89. Ecrire un programme qui affiche les valeurs de adr1, adr2 et de leur contenu.

5.5 AFFECTATION D'UNE VALEUR A UN POINTEUR

Dans les exemples précédents, l'utilisateur **ne choisit pas** la valeur des adresses mémoire **attribuées par le compilateur à chaque variable.** L'utilisateur se contente de **lire** la valeur de ces adresses en l' affichant sur l'écran.

On ne peut pas affecter directement une valeur à un pointeur:

l'écriture suivante est interdite: **char *pc;**

pc = 0xfffe;

On peut cependant être amené à **définir par programmation** la valeur d'une adresse. On utilise pour cela l'opérateur de "cast", jeu de deux parenthèses.

- Par exemple pour adresser un périphérique (adressage physique),

- Par exemple pour contrôler la zône DATA dans un plan mémoire.

Exemples:

```
char *pc;

pc = (char*)0x1000;   /* p est l'adresse 0x1000 et pointe sur un caractère */

int *pi;

pi = (int*)0xfffa;        /* i est l'adresse 0xfffa et pointe sur un entier */
```

Lorsqu'on utilise une fonction d'allocation dynamique on ne peut affecter de valeur au pointeur à l'aide de l'opérateur de cast.

5.6 LES TABLEAUX DE NOMBRES (INT ou FLOAT)

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

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 pour 10 entiers, soit 20 octets en TURBOC et 40 octets en C standard.

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

Remarque: dim est nécessairement une VALEUR NUMERIQUE. Ce ne peut être en aucun cas une combinaison des variables du programme.

Utilisation: Un élément du tableau est repéré par son indice. En langage C les tableaux commencent à l'indice 0. L'indice maximum est donc dim-1.

Appel: **nom[indice]**

Exemples: **compteur[2] = 5;**

nombre[i] = 6.789;

Tous les éléments d'un tableau (correspondant au dimensionnement maximum) ne sont pas forcément définis.

D'une façon générale, les tableaux consomment beaucoup de place en mémoire. On a donc intérêt à les dimensionner au plus juste.

Exercice VI_1: Saisir 10 réels, les ranger dans un tableau. Calculer et afficher la moyenne et l'écart-type.

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 maximum sont donc dim1-1, dim2-1.

Appel: **nom[indice1][indice2]**

Exemples: **compteur[2][4] = 5;**

nombre[i][j] = 6.789;

Tous les éléments d'un tableau (correspondant au dimensionnement maximum) ne sont pas forcément définis.

5.7 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,-8,0.09};

int x[2][3] = {{1,5,7},{8,4,3}}; /* 2 lignes et 3 colonnes */

En déclarant un tableau, on définit automatiquement un pointeur (on définit en fait l'adresse du premier élément du tableau).

5.8 Les tableaux à une dimension:

Les écritures suivantes sont équivalentes:

int *tableau;	**int tableau[10];**	**déclaration**
tableau = (int*)malloc(sizeof(int)*10);		
***tableau**	**tableau[0]**	**le 1er élément**
***(tableau+i)**	**tableau[i]**	**un autre élément**
tableau	**&tableau[0]**	**adresse du 1er élément**
(tableau + i)	**&(tableau[i])**	**adresse d'un autre élément**

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

Remarque: La déclaration d'un tableau entraine automatiquement la réservation de places en mémoire. Ce n'est pas le cas lorsque l'on déclare un pointeur. Il faut alors utiliser une fonction d'allocation dynamique comme malloc.

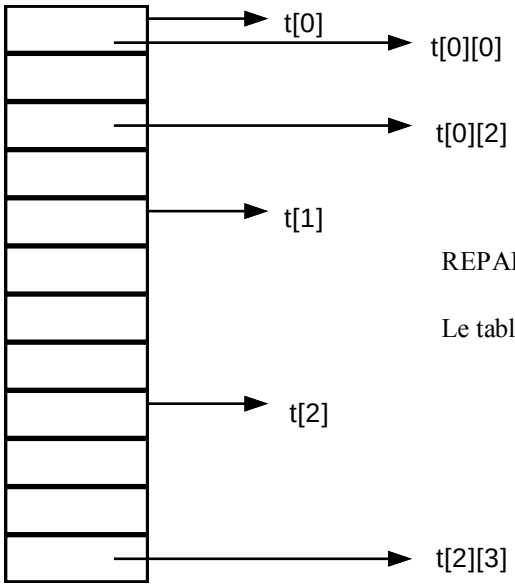
5.9 Les tableaux à plusieurs dimensions:

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



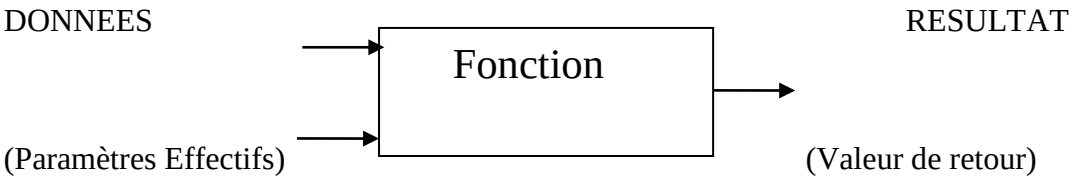
REPARTITION DES ELEMENTSEN MEMOIRE

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

CHAPITRE 6
LES FONCTIONS

En langage C++ les sous-programmes s'appellent des **fonctions.** L'imbrication de fonctions n'est pas autorisée en C++: une fonction ne peut pas être déclarée à l'intérieur d'une autre fonction. Par contre, une fonction peut **appeler** une autre fonction. Cette dernière doit être déclarée **avant** celle qui l'appelle.

Une fonction au sens strict peut être représentée par une boite noire (un mécanisme invisible) qui donne un et un seul résultat à partir d'une ou plusieurs données (voire aucune).



Le résultat d'une fonction est appelée VALEUR DE RETOUR. Les données à partir desquelles une fonction calcule son résultat sont appelées PARAMETRES EFFECTIFS ou ARGUMENTS

Une variable connue uniquement d'une fonction ou de main() est une variable locale.

Une variable connue de tout le programme est une variable globale.

En C++, la notion de fonction est plus large : en C++, une fonction peut ne rien retourner. En C++, on utilise le mot fonction pour désigner tous les sous-programmes, y compris les procédures.

6.1 Utilisation des fonctions standards

En C++, les fonctions standards sont regroupées dans des bibliothèques (library en anglais). Pour utiliser une fonction standard, il faut inclure le fichier d'en-tête où elle est déclarée. Par exemple, pour utiliser la fonction getch( ), qui se contente de saisir sans retour à l'écran d'un caractère tapé au clavier, il faut inclure le fichier d'en-tête .<conio.h>

Ensuite, pour exécuter une fonction à l'intérieur d'un programme, il faut effectuer un APPEL de cette fonction. L'appel d'une fonction consiste tout simplement à écrire son nom, suivi entre parenthèses des paramètres effectifs (les données).

Exemple :

Voilà un programme qui permet tout simplement d'afficher la racine carrée d'un nombre entier saisi par l'utilisateur. Pour cela, on utilise la fonction sqrt déclarée dans le fichier d'en-tête math.h.

Exemple

```
#include <iostream>
#include <math.h>
 main( )
{ int n;
  cout << "Taper un nombre ";
  cin >> n;
cout << "La racine carrée de ce nombre est "<< sqrt(n) ; }
```

A l'exécution, l'appel d'une fonction est remplacé par la valeur retournée (résultat) : sqrt(n) correspond à la valeur de la racine carrée de n. Donc on peut utiliser l'appel d'une fonction comme toute autre valeur : on peut l'afficher, l'affecter à une variable ou l'utiliser dans une expression (calcul, condition,…).

II. La création de fonctions

En vertu des principes de la programmation modulaire, qui préconise de découper le code des applications en petites unités, les programmeurs sont conduits à créer leurs propres sousprogrammes. Nous allons voir comment faire en C++.

Spécificités du C et C++:

• **tous les sous-programmes sont appelés FONCTIONS** même ceux qui ne renvoie rien (et qu'on appelle procédure en algorithmique et dans les autres langages).

• **le programme principal est lui même une fonction**, appelée obligatoirement main( ) car c'est la toute première fonction appelée à l'exécution du programme.

Définition et appel d'une fonction (au sens algorithmique)

- **Définition**

Syntaxe:

```
type_retourné nom_fonction (type_paramètre nom_paramètre, …) //en-tête
{
```

```
/*corps de la fonction*/
return valeur_retournée;
}
```

Exemple: définition de la fonction somcarre ( )

```
double somcarre (float a, float b)
{
    double sc;
    sc = a * a + b * b;
    return sc; }
```

ou plus simplement

```
double somcarre (float a, float b)
{
    return a*a + b*b;
}
```

L'appel d'une fonction

L'appel d'une fonction est utilisé dans une instruction comme une valeur. On peut même utiliser l'appel d'une fonction comme valeur de paramètre pour une autre fonction.

ex: appel de la fonction somcarre pour calculer la variance v de deux nombres n1 et n2.

```
double v;
float n1, n2; …//saisie de n1 et n2
v = (somcarre(n1, n2) – (n1+n2)) / n;
```

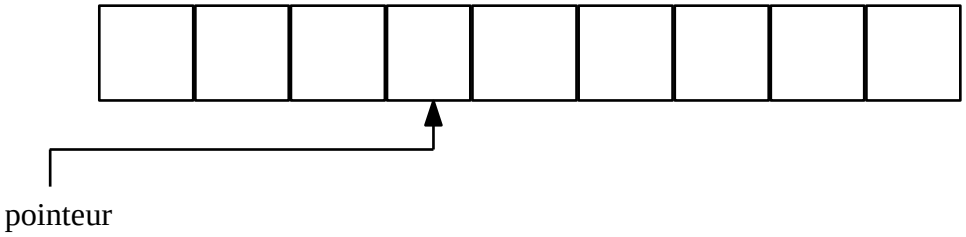
Remarquez que les types des paramètres effectifs correspondent au type des paramètres formels correspondants.

CHAPITRE 7
LES FICHIERS

7.1 GENERALITES

En C, les communications d'un programme avec son environnement se font par l'intermédiaire de fichiers. Un fichier (angl.: file) est un ensemble structuré de données stocké en général sur un support externe (disquette, disque dur, disque optique, bande magnétique, ...). Un fichier structuré contient une suite d'enregistrements homogènes, qui regroupent le plus souvent plusieurs composantes appartenant ensemble (champs). En C, un fichier est une suite d'octets. Les informations contenues dans le fichier ne sont pas forcément de même type (un char, un int, une structure ...)

Un **pointeur** fournit l'adresse d'une information quelconque.



On distingue généralement deux types d'accès:

1- Accès séquentiel (possible sur tout support, mais seul possible sur bande magnétique):

- Pas de cellule vide.

- On accède à une cellule quelconque en se déplaçant (via un pointeur), depuis la cellule de départ.

- On ne peut pas détruire une cellule.

- On peut par contre tronquer la fin du fichier.

- On peut ajouter une cellule à la fin.

2- Accès direct (RANDOM I/O) (Utilisé sur disques, disquettes, CD-ROM où l'accès séquentiel est possible aussi).

- Cellule vide possible.

- On peut directement accéder à une cellule.

- On peut modifier (voir détruire) n'importe quelle cellule.

Il existe d'autre part deux façons de coder les informations stockées dans un fichier :

1- En binaire :

Fichier dit « binaire », les informations sont codées telles que. Ce sont en général des fichiers de nombres. Ils ne sont pas listables.

2- en ASCII :

Fichier dit « texte », les informations sont codées en ASCII. Ces fichiers sont listables. Le dernier octet de ces fichiers est EOF (caractère ASCII spécifique).

7.2 MANIPULATION DES FICHIERS

Il existe deux fichiers spéciaux qui sont définis par défaut pour tous les programmes:

- ***stdin*** le fichier d'entrée standard

- ***stdout*** le fichier de sortie standard

En général, stdin est lié au clavier et stdout est lié à l'écran, c.-à-d. les programmes lisent leurs données au clavier et écrivent les résultats sur l'écran. 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++.

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.

1 - Déclaration: **FILE *fichier; /* majuscules obligatoires pour FILE */**

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

La déclaration des fichiers doit figurer AVANT la déclaration des autres variables.

2 - Ouverture: **FILE *fopen(char *nom, char *mode);**

On passe donc 2 chaînes de caractères

nom: celui figurant sur le disque, 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+ »)

Exemple : **FILE *fichier ;**

fichier = fopen(« a :\toto.dat », « rb ») ;

3 - Fermeture: **int fclose(FILE *fichier);**

Retourne 0 si la fermeture s'est bien passée, EOF en cas d'erreur.

Il faut toujours fermer un fichier à la fin d'une session. mode (pour les fichiers TEXTE) :

Exemple : **FILE *fichier ;**

fichier = fopen(« a :\toto.dat », « rb ») ;

/* Ici instructions de traitement */

fclose(fichier) ;

4 - Destruction: **int remove(char *nom);**

Retourne 0 si la fermeture s'est bien passée.

Exemple : **remove(« a :\toto.dat ») ;**

5 - Renommer: **int rename(char *oldname, char *newname);**

Retourne 0 si la fermeture s'est bien passée.

6 - Positionnement du pointeur au début du fichier: **void rewind(FILE *fichier);**

7- Ecriture dans le fichier:

int putc(char c, FILE *fichier);

Ecrit la valeur de c à la position courante du pointeur , le pointeur avance d'une case mémoire.

Retourne EOF en cas d'erreur.

Exemple : **putc('A', fichier) ;**

int putw(int n, FILE *fichier);

Idem, n de type int, le pointeur avance du nombre de cases correspondant à la taille d'un entier (4 cases en C standard).

Retourne n si l'écriture s'est bien passée.

int fputs(char *chaîne, FILE *fichier); idem avec une chaîne de caractères, le pointeur avance de la longueur de la chaine ('\0' n'est pas rangé dans le fichier).

Retourne EOF en cas d'erreur.

Exemple : **fputs(« BONJOUR ! », fichier) ;**

int fwrite(void *p,int taille_bloc,int nb_bloc,FILE *fichier); p de type pointeur, écrit à partir de la position courante du pointeur fichier 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.

Retourne le nombre de blocs écrits.

Exemple: taille_bloc=4 (taille d'un entier en C), nb_bloc=3, écriture de 3 octets.

```
int tab[10] ;

fwrite(tab,4,3,fichier) ;
```

int fprintf(FILE *fichier, char *format, liste d'expressions); réservée plutôt aux fichiers ASCII.

Retourne EOF en cas d'erreur.

Exemples: **fprintf(fichier,"%s","il fait beau");**

```
fprintf(fichier,%d,n);

fprintf(fichier,"%s%d","il fait beau",n);
```

Le pointeur avance d'autant.

8 - Lecture du fichier:

int getc(FILE *fichier); lit 1 caractère, mais retourne un entier n; retourne EOF si erreur ou fin de fichier; le pointeur avance d'une case.

Exemple: **char c ;**

```
c = (char)getc(fichier) ;
```

int getw(FILE *fichier); idem avec un entier; le pointeur avance de la taille d'un entier.

Exemple: **int n ;**

```
n = getw(fichier) ;
```

char *fgets(char *chaine,int n,FILE *fichier); lit n-1 caractères à partir de la position du pointeur et les range dans chaine en ajoutant '\0'.

int fread(void *p,int taille_bloc,int nb_bloc,FILE *fichier); analogue à fwrite en lecture.

Retourne le nombre de blocs lus, et 0 à la fin du fichier.

int fscanf(FILE *fichier, char *format, liste d'adresses); analogue à fprintf en lecture.

9 - Gestion des erreurs:

fopen retourne le pointeur NULL si erreur (Exemple: impossibilité d'ouvrir le fichier).

fgets retourne le pointeur NULL en cas d'erreur ou si la fin du fichier est atteinte.

la fonction **int feof(FILE *fichier)** retourne 0 tant que la fin du fichier n'est pas atteinte.

la fonction **int ferror(FILE *fichier)** retourne 1 si une erreur est apparue lors d'une manipulation de fichier, 0 dans le cas contraire.

10 - Fonction particulière aux fichiers à acces direct:

int fseek(FILE *fichier,int offset,int direction) déplace le pointeur de offset cases à partir de direction.

Valeurs possibles pour direction:

0 -> à partir du début du fichier.

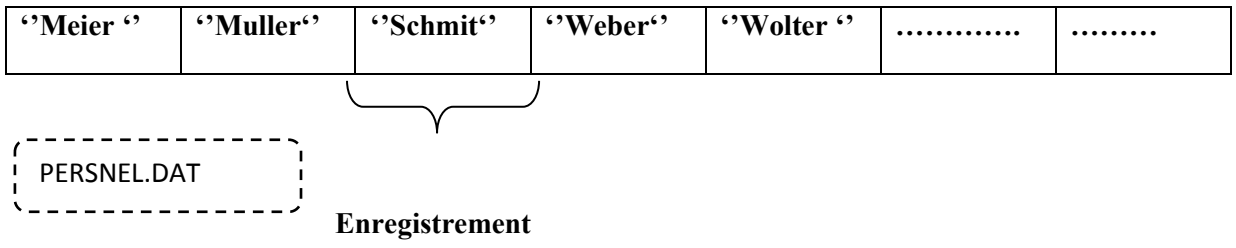
1 -> à partir de la position courante du pointeur.

2 -> en arrière, à partir de la fin du fichier.

Retourne 0 si le pointeur a pu être déplacé;

7.3 Problème

On se propose de créer un fichier qui est formé d'enregistrements contenant comme information le nom d'une personne. Chaque enregistrement est donc constitué d'une seule rubrique, à savoir, le nom de la personne.



L'utilisateur doit entrer au clavier le nom du fichier, le nombre de personnes et les noms des personnes. Le programme se chargera de créer le fichier correspondant sur disque dur ou sur disquette.

Après avoir écrit et fermé le fichier, le programme va rouvrir le même fichier en lecture et afficher son contenu, sans utiliser le nombre d'enregistrements introduit dans la première partie.

Solution en langage algorithmique

```
programme PERSONNEL
  chaîne NOM_FICHIER, NOM_PERS
  entier C,_NB_ENREG

  (* Première partie :
     Créer et remplir le fichier *)
  écrire "Entrez le nom du fichier à créer : "
  lire NOM_FICHIER
  ouvrir NOM_FICHIER en écriture
   écrire "Nombre d'enregistrements à créer : "
  lire NB_ENREG
  en C ranger 0
  tant que (C<NB_ENREG) faire
  |  écrire "Entrez le nom de la personne : "
  |  lire NOM_PERS
  |  écrire NOM_FICHIER:NOM_PERS
  |  en C ranger C+1
  ftant (* C=NB_ENREG *)
  fermer NOM_FICHIER
(* Deuxième partie :
     Lire et afficher le contenu du fichier *)
  ouvrir NOM_FICHIER en lecture
  en C ranger 0
  tant que non(finfichier(NOM_FICHIER)) faire
  |  lire NOM_FICHIER:NOM_PERS
  |  écrire "NOM : ",NOM_PERS
  |  en C ranger C+1
  ftant
  fermer NOM_FICHIER
fprogramme (* fin PERSONNEL *)
```

Solution en langage C

```
#include <stdio.h>

main()
{
  FILE *P_FICHIER; /* pointeur sur FILE */
  char NOM_FICHIER[30], NOM_PERS[30];
  int  C,NB_ENREG;

  /* Première partie :
     Créer et remplir le fichier */
  printf("Entrez le nom du fichier à créer : ");
```

```
  scanf("%s", NOM_FICHIER);
  P_FICHIER = fopen(NOM_FICHIER, "w");  /* write */
  printf("Nombre d'enregistrements à créer : ");
  scanf("%d", &NB_ENREG);
  C = 0;
  while (C<NB_ENREG)
     {
     printf("Entrez le nom de la personne : ");
     scanf("%s", NOM_PERS);
     fprintf(P_FICHIER, "%s\n", NOM_PERS);
     C++;
     }
  fclose(P_FICHIER);

  /* Deuxième partie :
     Lire et afficher le contenu du fichier */
  P_FICHIER = fopen(NOM_FICHIER, "r");  /* read */
  C = 0;
  while (!feof(P_FICHIER))
     {
     fscanf(P_FICHIER, "%s\n", NOM_PERS);
     printf("NOM : %s\n", NOM_PERS);
     C++;
     }
  fclose(P_FICHIER);
  return 0;
}
```

CHAPITRE 9
PROGRAMMATION ORIENTEE OBJET EN C++

Apparue au début des années 70, la programmation orientée objet répond aux nécessites de l'informatique professionnelle. Elle offre aux concepteurs de logiciels une grande souplesse de travail, permet une maintenance et une évolution plus aisée des produits. Mais sa pratique passe par une approche radicalement différente des méthodes de programmation traditionnelles : avec les langages à objets, le programmeur devient metteur en scène d'un jeu collectif où chaque objet-acteur se voit attribuer un rôle bien précis. Ce cours a pour but d'expliquer les règles de ce jeu. La syntaxe de base du langage C++.

9.1 Problématique de la programmation

Concevoir des logiciels avec des exigences de qualité :

Exactitude : aptitude d'un logiciel à fournir les résultats voulus dans des conditions normales d'utilisation. Robustesse : aptitude à bien réagir quand on s'écarte des conditions normales d'utilisation.
Extensibilité : facilité avec laquelle un programme pourra être adapte pour satisfaire à l´évolution de la demande.
Réutilisabilité : possibilité d'utiliser certaines parties du logiciel pour résoudre d'autres problèmes.
Portabilité : facilité avec laquelle on peut exploiter un même logiciel dans différentes implémentations.
Efficience : temps d'exécution, taille mémoire ...

Repose sur" l´équation de Writh" :

Algorithmes + Structures de données = Programmes

Les algorithmes étant dissocient des données, il est nécessaire d'établir le lien entre eux lors de la conception du programme. La modification du logiciel par la remise en cause de l'un des éléments nécessite souvent la remise en cause de l'ensemble des éléments.

Repose sur le concept d'objet qui est une association de données et de procédures (ou méthodes) agissant sur ces données :

Méthodes + Données = Objet

Les classes

Elles généralisent les structures en C.

Elles constituent un modèle ou type abstrait permettant de déclarer des instances (objets) ayant les caractéristiques de la classe.

Une classe contient :

des données-membres ou attributs.

des fonctions-membres ou méthodes

Chaque membre se voit définir un droit d'accès par l'un des mots-clés public, private ou protected.

Seuls les membres publics sont accessibles de l'extérieur de la classe. Ceci renforce l'encapsulation des données.

Les membres publics constituent l'interface pour l'utilisateur

Squelette d'une classe

```
class Nom_class {
private:
// déclaration des attributs
// et méthodes privées
 protected:
// Déclaration des attributs
 // et méthodes protèges
public:
// Déclaration des attributs
 // et méthodes publics
};
```

Exemple : la classe Point

```
class Point {
 private:
int coord_x;
int coord_y;
public:
void SetCoord(const  int  & x, const int  & y);
void Deplacer(const int & dx, const int & dy);
```

void Afficher() const; };

Les constructeurs

Avant de pouvoir être utilisé, un objet doit être construit. Cette opération est effectuée par des m´méthodes spécifiques : les constructeurs. On peut en avoir plusieurs pour une même classe

Ils portent toujours le nom de la classe.

Ils ne retournent aucun résultat, pas même void.

Si aucun constructeur n'est d´déclare, un constructeur par défaut, sans paramètres, est automatiquement crée. Ce constructeur fera appel aux constructeurs par défaut des attributs qui composent l'objet.

Exmple : la classe point

Fichier Point.hpp

class Point {

private:

int coord_x;

int coord_y;

public:

Point();

Point(const in & x, const int & y);

Point(const Point & p);

void SetCoord(const int & x, const int & y);

void Deplacer(const int & dx, const int & dy);

void Afficher() const; };

Le destructeur

Lorsqu'un objet devient inutile, il doit être détruit. Cette opération est réalisée par le destructeur. Caractéristiques :

Il est unique.

Son nom est le nom de la classe précède par un ˜ .

Il n'a pas de paramètre.

Il ne retourne aucun résultat, pas même void.

Il doit restituer la place mémoire allouée dynamiquement par l'objet.

Exemple : la classe Point

Fichier Point.hpp :

```
class Point {
private:
int coord_x;
int coord_y;
public:
Point();
Point(const in & x, const int & y);
 Point(const Point & p);
~Point();
void SetCoord(const int & x, const int & y);
void Deplacer(const int & dx, const int & dy);
void Afficher()
const; };
```

Utilisation des objets

```
#include "Point.hpp"
int main(){
Point p1; // Constructeur par defaut
 Point p2(5,3); // Constructeur avec parametre
Point p3(p1); // Constructeur de copie
Point p4 = p2; // Constructeur de copie
p2.Afficher(); // Affiche le point p2
 p1.SetCoord(7,2); // Modifie les coordonnées de p1
Point * pt1; // Déclaration d'un pointeur sur un Point
 pt1 = new Point; // Allocation dynamique - Const. par defaut
 Point * pt2; // Déclaration d'un pointeur sur un Point
pt2 = new Point(3,1) // Allocation dynamique - Const. avec param
```

```
(*pt1).Afficher() // Affiche le point pointe par pt1;

pt1->Afficher() // Idem instruction precedente

delete pt1; // Appel destructeur pour *pt1

delete pt2; // Appel destructeur pour *pt2

 return 0; } // Appel destructeur pour p1, p2, p3 et p4
```

Réalisation des fonctions-membres

On écrit généralement les fonctions-membres en dehors de la classe dans un fichier .cpp. On utilise alors l'operateur :: de réalisation de domaine pour préfixer le nom de la classe au nom de la méthode.

A l'intérieur de la fonction-membre, on travaille toujours par défaut sur l'instance "acteur" courante (celui-ci correspond à l'objet sur lequel on a appelé la méthode). Les membres (attributs et méthodes) de cet objet sont accessibles directement par leur nom, sans préfixe. Les membres des autres objets de la même classe, sont accessibles par la notation pointée.

Exemple : la classe Point

Fichier Point.cpp :

```
 #include "Point.hpp" Point::Point(){}coord_x = 0;

coord_y = 0; }

Point::Point(const in & x, const int & y): coord_x(x),

coord_y(y){ }

Point::Point(const Point & p){

coord_x = p.coord_x;

 coord_y = p.coord_y; }

Point::~Point(){}

void Point::SetCoord(const int & x, const int & y){ coord_x = x;

coord_y = y; }

 void Point::Deplacer(const int & dx, const int & dy){

coord_x += dx; // coord_x = coord_x + dx; coord_y += dy; // coord_y = coord_y + dy; }

void Point::Afficher(){

cout << "x = " << coord_x << " y = " << coord_y << endl; }
```

Travaux pratiques

Travaux pratiques (TP°1)

Exercice 1

Éliminer les parenthèses superflues dans les expressions suivantes :
a = (x+5) /* expression 1 */
a = (x=y) + 2 /* expression 2 */
a = (x==y) /* expression 3 */
(a<b) && (c<d) /* expression 4 */
(i++) * (n+p) /* expression 5 */

Exercice 2 :

Quels résultats fournit le programme suivant :

```
// my first program in C++
#include <iostream>
int main()
{
  std::cout << "Hello World!";
}
```

Exercice 3:

Modifier Le programme « Hello World » de façon à obtenir le même résultat sur l'écran en utilisant plusieurs fois la fonction cout

Exercice 4

Quels résultats fournit le programme suivant :

```
#include <iostream>
using namespace std ;
main()
{
int n = 25 ; long p = 250000; unsigned q = 63000 ;
char c = 'a' ;
float x = 12.3456789 ; double y = 12.3456789e16 ;
char * ch = "bonjour" ;
int * ad = & n ;
cout << "valeur de n : " << n << "\n" ;
cout << "valeur de p : " << p << "\n" ;
cout << "caractere c : " << c << "\n" ;
cout << "valeur de q : " << q << "\n" ;
cout << "valeur de x : " << x << "\n" ;
cout << "valeur de y : " << y << "\n" ;
cout << "chaine ch : " << ch << "\n" ;
cout << "adresse de n : " << ad << "\n" ;
cout << "adresse de ch : " << (void *) ch << "\n" ;
}
```

Exercice 5

Quels résultats fournit le programme suivant :

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

```
{
int i, j, n ;
 i = 0 ; n = i++ ;
cout << "A : i = " << i << " n = " << n << "\n" ;
 i = 10 ; n = ++ i ;
cout << "B : i = " << i << " n = " << n << "\n" ;
i = 20 ; j = 5 ; n = i++ * ++ j ;
 cout << "C : i = " << i << " j = " << j << " n = " << n << "\n" ;
 i = 15 ; n = i += 3 ;
 cout << "D : i = " << i << " n = " << n << "\n" ;
i = 3 ; j = 5 ; n = i *= --j ;
cout << "E : i = " << i << " j = " << j << " n = " << n << "\n" ;
}
```

Exercice 6

Quels résultats fournira ce programme :

```
#include <iostream>
using namespace std ;
main()
{
 int n=10, p=5, q=10, r ;
 r = n == (p = q) ;
cout << "A : n = " << n << " p = " << p << " q = " << q << " r = " << r << "\n" ;
n = p = q = 5 ;
n += p += q ;
cout << "B : n = " << n << " p = " << p << " q = " << q << "\n" ;
q = n < p ? n++ : p++ ;
cout << "C : n = " << n << " p = " << p << " q = " << q << "\n" ; q = n > p ? n++ : p++ ;
cout << "D : n = " << n << " p = " << p << " q = " << q << "\n" ;
}
```

Exercice 7

Quels résultats fournira ce programme :

```
 #include <iostream>
using namespace std ;
main()
{
 int n, p, q ;
n = 5 ; p = 2 ;    /* cas 1 */
q = n++ >p || p++ != 3 ;
cout << "A : n = " << n << " p = " << p << " q = " << q << "\n" ;
n = 5 ; p = 2 ; /* cas 2 */
q = n++
cout << "B : n = " << n << " p = " << p << " q = " << q << "\n" ;
n = 5 ; p = 2 ; /* cas 3 */
 q = ++n == 3 && ++p == 3 ;
cout << "C : n = " << n << " p = " << p << " q = " << q << "\n" ;
n = 5 ; p = 2 ; /* cas 4 */
q = ++n == 6 && ++p == 3 ;
cout << "D : n = " << n << " p = " << p << " q = " << q << "\n" ; }
```

Travaux pratiques (TP°2)

Exercice 1

Soit le programme suivant :

```
#include <iostream>
int main()
{
 int n;
        std::cin>> n ;
switch (n)
{ case 0 :  std::cout << "Nul\n" ;
case 1 :
case 2 :  std::cout << "Petit\n" ;
break ;
case 3 :
case 4 :
case 5 :  std::cout << "Moyen\n" ;
default :  std::cout << "Grand\n" ;
}
system("PAUSE");
}
```

Quels résultats affiche-t-il lorsqu'on lui fournit en donnée :
a. 0, b. 1 ;c. 4 ;d. 10 ;e. -5

Exercice 2

Quels résultats fournit le programme suivant :

```
#include <iostream>
using namespace std ;
main()
{ int n, p ;
n=p=0 ;
while (n<5) n+=2 ; p++ ;
cout << "A : n = " << n << " p = " << p << "\n" ;
n=p=0 ;
while (n<5) { n+=2 ; p++ ; }
cout << "B : n = " << n << " p = " << p << "\n" ;
}
```

Exercice 3

Écrire un programme qui calcule les racines carrées de nombres fournis en donnée. Il s'arrêtera
lorsqu'on lui fournira la valeur 0. Il refusera les valeurs négatives. Son exécution se présentera ainsi :
donnez un nombre positif : 2
sa racine carrée est : 1.414214e+00
donnez un nombre positif : -1
svp positif
donnez un nombre positif : 5
sa racine carrée est : 2.236068e+00
donnez un nombre positif : 0

Travaux pratiques (TP°3) (Les instructions de contrôle)

Exercice 1

Quels résultats fournit le programme suivant :

```
#include <iostream>
using namespace std ;
main()
{ int n=0 ;
do
{ if (n%2==0) { cout << n << " est pair\n" ;
n += 3 ;
continue ;
}
if (n%3==0) { cout << n << " est multiple de 3\n" ;
n += 5 ;
}
if (n%5==0) { cout << n << " est multiple de 5\n" ;
break ;
}
n += 1 ;
}
while (1) ;
system("PAUSE");}
```

Exercice 2

Quels résultats fournit le programme suivant :

```
#include <iostream>
using namespace std ;
main()
{ int n, p ;
n=p=0 ;
while (n<5) n+=2 ; p++ ;
cout << "A : n = " << n << " p = " << p << "\n" ;
n=p=0 ;
while (n<5) { n+=2 ; p++ ; }
cout << "B : n = " << n << " p = " << p << "\n" ;
}
```

Exercice 3

Soit le petit programme suivant :

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

Exercice 4

Quels résultats fournit le programme suivant :

```
#include <iostream>
using namespace std ;
main()
{ int i, n ;
for (i=0, n=0 ; i<5 ; i++) n++ ;
cout << "A : i = " << i << " n = " << n << "\n" ;
for (i=0, n=0 ; i<5 ; i++, n++) {}
cout << "B : i = " << i << " n = " << n << "\n" ;
for (i=0, n=50 ; n>10 ; i++, n-= i ) {}
cout << "C : i = " << i << " n = " << n << "\n" ;
for (i=0, n=0 ; i<3 ; i++, n+=i,
cout << "D : i = " << i << " n = " << n << "\n" ) ;
cout << "E : i = " << i << " n = " << n << "\n" ;
}
```

Travaux pratiques (TP°4) (Tableaux et pointeurs)

Exercice 1

Quels résultats fournira ce programme :

```
#include <stdio.h>
#include <iostream>
using namespace std ;
main()
{
int t [3] ;
int i, j ;
int * adt ;
for (i=0, j=0 ; i<3 ; i++) t[i] = j++ + i ; /* 1 */
for (i=0 ; i<3 ; i++) cout << t[i] << " " ; /* 2 */
cout << "\n" ;
for (i=0 ; i<3 ; i++) cout << *(t+i) << " " ; /* 3 */
printf ("\n") ;
for (adt = t ; adt < t+3 ; adt++) cout << *adt << " " ; /* 4 */
cout << "\n" ;
for (adt = t+2 ; adt>=t ; adt--) cout << *adt << " "; /* 5 */
cout << "\n" ;
}
```

Exercice 2

Écrire, de deux façons différentes, un programme qui lit 10 nombres entiers dans un tableau avant d'en rechercher le plus grand et le plus petit :
a. en utilisant uniquement le « formalisme tableau » ;
b. en utilisant le « formalisme pointeur », à chaque fois que cela est possible.

Exercice 3

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 le formalisme tableau.

Travaux pratiques (TP°5) (Fonctions)

Exercice 1

Quelle modification faut-il apporter au programme suivant pour qu'il devienne correct :

```
#include <iostream>
using namespace std ;
main()
{ int n, p=5 ;
n = fct (p) ;
cout << "p = " << p << " n = " << n ;
}
int fct (int r)
{ return 2*r ;
}
```

Exercice 2

Écrire :

- une fonction, nommée f1, se contentant d'afficher « bonjour » (elle ne possédera aucun argument, ni valeur de retour) ;
- une fonction, nommée f2, qui affiche « bonjour » un nombre de fois égal à la valeur reçue en argument (int) et qui ne renvoie aucune valeur ;
- une fonction, nommée f3, qui fait la même chose que f2, mais qui, de plus, renvoie la valeur (int) 0.

Écrire un petit programme appelant successivement chacune de ces 3 fonctions, après les avoir convenablement déclarées (on ne fera aucune hypothèse sur les emplacements relatifs des différentes fonctions composant le fichier source).

Travaux pratiques (TP°6) (Programmation orientée objet en C++**)**

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.

Références bibliographiques:

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

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

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

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

[5] Claude Delannoy, « Programmer en langage C++ », Edition Eyrolles, 2000.

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

[7] Bjarne Stroustrup, « Le Langage C++ », Édition Addison-Wesley, 2000.