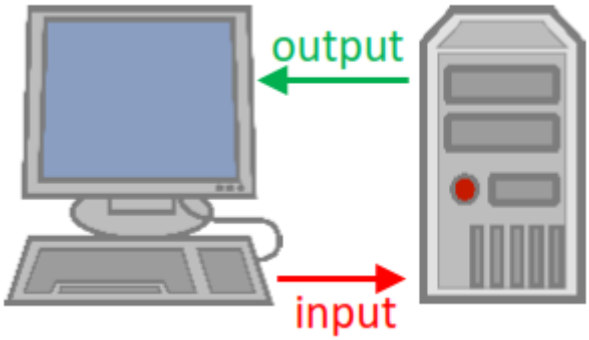
Université Hassiba Benbouali de Chlef
Département d'Electronique

Faculté de Technologie
Filière : Automatique

TRAVAUX PRATIQUES PROGRAMMATION C++



Dr. TAHRAOUI SOUAD

ANNEE UNIVERSITAIRE 2023/2024

Avant-propos

Ce polycopie est constitué de textes de travaux pratiques (TP) élabores pour étudier pratiquement la programmation en langage C++. L'accent y est mis sur l'utilisation pratique du logiciel **Code::Blocks**. Ces TP sont destinés aux étudiants de troisième année licence en automatique.

Dans cette perspective, les textes de travaux pratiques présentes dans ce polycopie sont constitués d'un rappel théorique suivi d'une série des applications. Ces applications permettent à l'étudiant d'apprendre la programmation en langage C++ . Ces travaux pratiques présentes dans ce polycopie se déroulent sur un semestre.

Semestre: 5

Unité d'enseignement: UEM 3.1

Matière 2: TP Programmation en C++

VHS: 15h00 (TP : 1h00)

Crédits: 1

Coefficient: 1

Objectifs de l'enseignement:

Ce module permettra à l'étudiant la mise en pratique et la consolidation des connaissances acquises dans le module de programmation en C++.

Connaissances préalables recommandées:

Module programmation en C++

Contenu de la matière:

TP 1: Familiarisation avec le langage C++

(Environnement de développement, compilation, débogage, exécution …)

TP 2: Syntaxe élémentaire, déclaration des variables et opérateurs

TP 3: Structures conditionnelles et les boucles

TP 4: Tableaux et pointeurs

TP 5: Fonctions

TP 6: Fichiers

TP 7: Programmation orientée objet en C++

Classes, Méthodes particulières (constructeurs, destructeurs…), Héritage

Mode d'évaluation:

Contrôle continu: 100%.

Note :

Les exercices ont été testés avec les outils **CODEBLOCK** en mode « **console** ».

Travaux Pratiques TP N°1

Familiarisation avec le langage C++

1-INTRODUCTION

Le langage **C++** est un des langages les plus célèbres au monde. Très utilisé, notamment dans le secteur des jeux vidéo qui apprécie ses performances et ses possibilités, le C++ est désormais incontournable pour les développeurs. Le C++ est le descendant du **langage C**. Ces deux langages, bien que semblables au premier abord, sont néanmoins *différents*. Le C++ propose de nouvelles fonctionnalités, comme la programmation orientée objet (POO). Elles en font un langage très puissant qui permet de programmer avec une approche différente du langage C.

Dans ce TP, le but est de introduire **Environnement de développement (programation C++), compilation, Débogage, Execution,……**

2-LES LOGICIELS NECESSAIRES POUR PROGRAMMER

Il faut installer certains logiciels spécifiques pour programmer en C++, 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

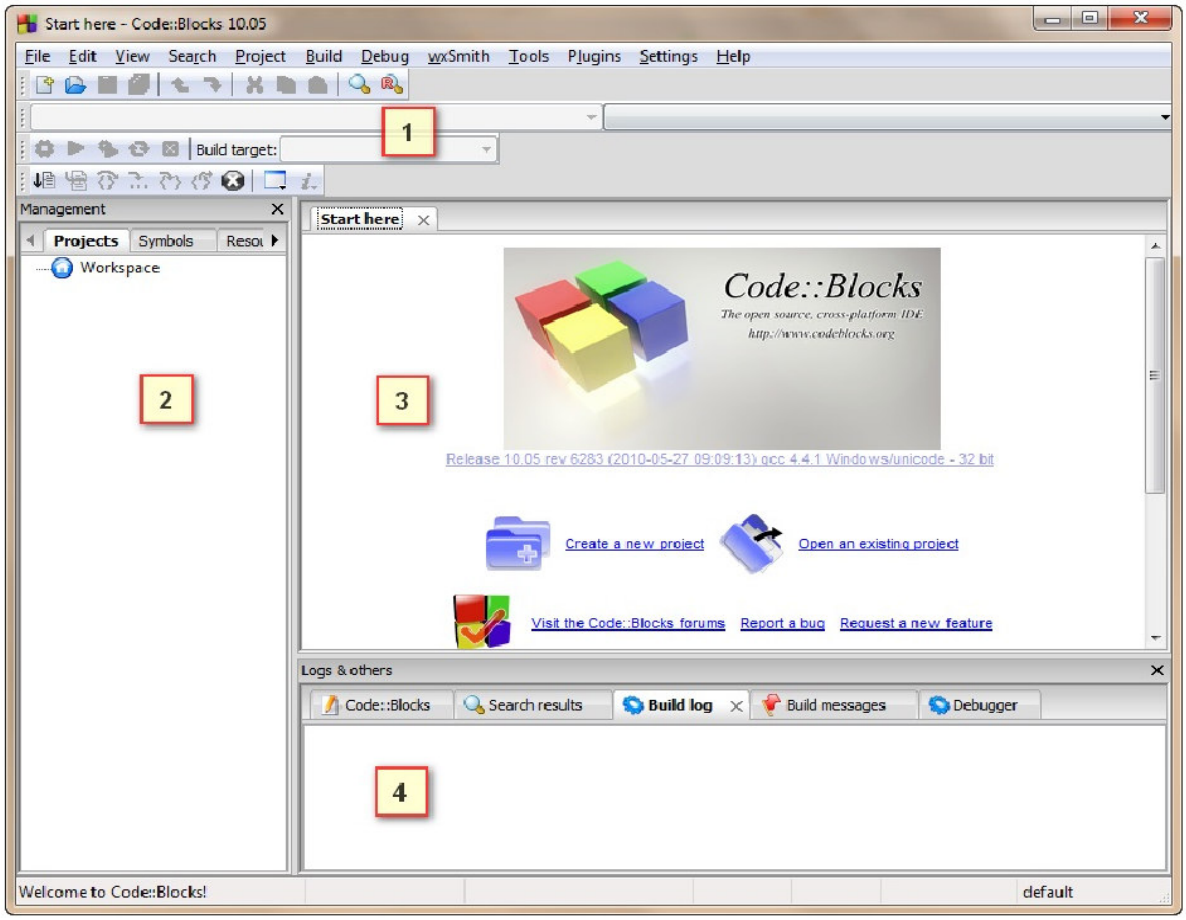
Xcode (Mac OS seulement)

Turbo C++

Code::blocks (Windows, Mac OS, Linux)

Nous avons découvert qu'il existait plusieurs compilateur (Code::Blocks, Visual C++, Xcode...). Nous travaillons sous **Code::Blocks.**.

Code::Blocks est un IDE libre et gratuit, disponible **pour Windows, Mac et Linux**.



On distingue 4 grandes sections dans la fenêtre, numérotées sur l'image :

1. **La barre d'outils** : elle comprend de nombreux boutons, mais seuls quelques-uns d'entre eux nous seront régulièrement utiles. J'y reviendrai plus loin.
2. **La liste des fichiers du projet** : c'est à gauche que s'affiche la liste de tous les fichiers source de votre programme. Notez que sur cette capture aucun projet n'a été créé donc on ne voit pas encore de fichiers à l'intérieur de la liste. Vous verrez cette section se remplir dans cinq minutes en lisant la suite du cours.
3. **La zone principale** : c'est là que vous pourrez écrire votre code en langage C++ !
4. **La zone de notification** : aussi appelée la "Zone de la mort", c'est ici que vous verrez les erreurs de compilation s'afficher si votre code comporte des erreurs. Cela arrive très régulièrement !

Intéressons-nous maintenant à une section particulière de la barre d'outils. Vous trouverez les boutons suivants (dans l'ordre) "Compiler", "Exécuter", "Compiler & Exécuter" et "Tout recompiler". Retenez-les, nous les utiliserons régulièrement.

1- **Compiler** : tous les fichiers source de votre projet sont envoyés au compilateur qui va se charger de créer un exécutable. S'il y a des erreurs (ce qui a de fortes chances d'arriver), l'exécutable ne sera pas créé et on vous indiquera les erreurs en bas de **Code::Blocks.**

2- **Exécuter** : cette icône lance juste le dernier exécutable que vous avez compilé. Cela vous permettra donc de tester votre programme et voir ainsi ce qu'il donne. Dans l'ordre, si vous avez bien suivi, on doit d'abord compiler, puis exécuter pour tester ce que ça donne. On peut aussi utiliser le 3ème bouton…

1. **Compiler & Exécuter** : pas besoin d'être un génie pour comprendre que c'est la combinaison des 2 boutons précédents.

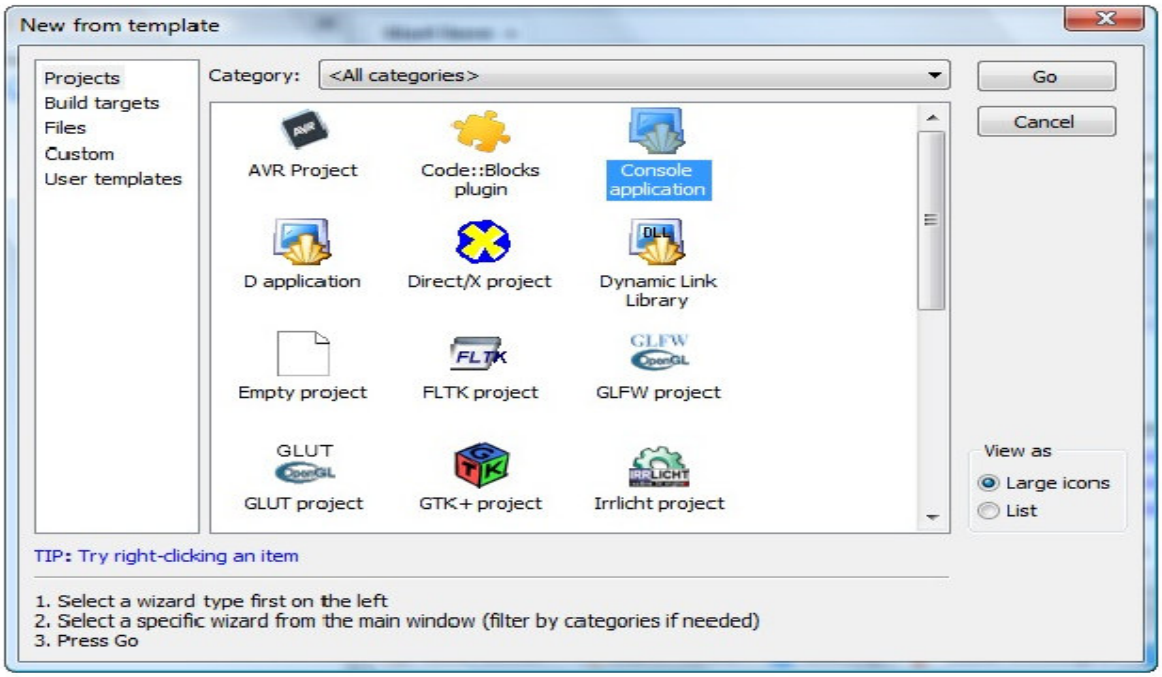
C'est d'ailleurs ce bouton que vous utiliserez le plus souvent. Notez que s'il y a des erreurs pendant la compilation (pendant la génération de l'exécutable), le programme ne sera pas exécuté. A la place, vous aurez droit à une belle liste d'erreurs à corriger.

1. **Tout reconstruire** : quand vous faites " Compiler ", **Code::Blocks** ne recompile en fait que les fichiers que vous avez modifiés et pas les autres. Parfois, je dis bien parfois, vous aurez besoin de demander à **Code::Blocks** de vous recompiler tous les fichiers. On verra plus tard quand on a besoin de ce bouton, et vous verrez plus en détail le fonctionnement de la compilation dans un chapitre futur. Pour l'instant, on se contente de savoir le minimum nécessaire pour pas tout mélanger. Ce bouton ne nous sera donc pas utile de suite.

3-CREATION D'UN NOUVEAU PROJET

Pour commencer à programmer, la première étape consiste à demander à son IDE (compilateur) de créer un nouveau projet. C'est un peu comme si vous demandiez à Word de vous créer un nouveau document.

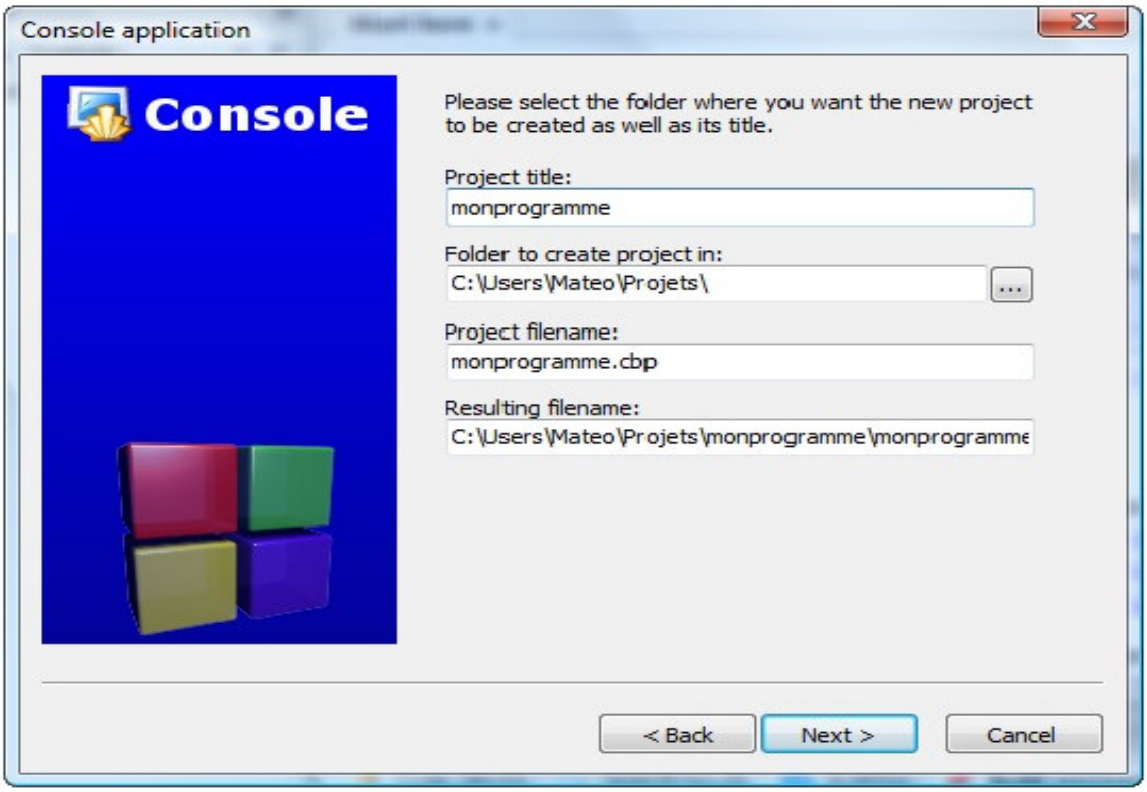
Pour créer un nouveau projet c'est très simple : allez dans le menu **File / New / Project.** Dans la fenêtre qui s'ouvre, choisissez "**Console application"** :



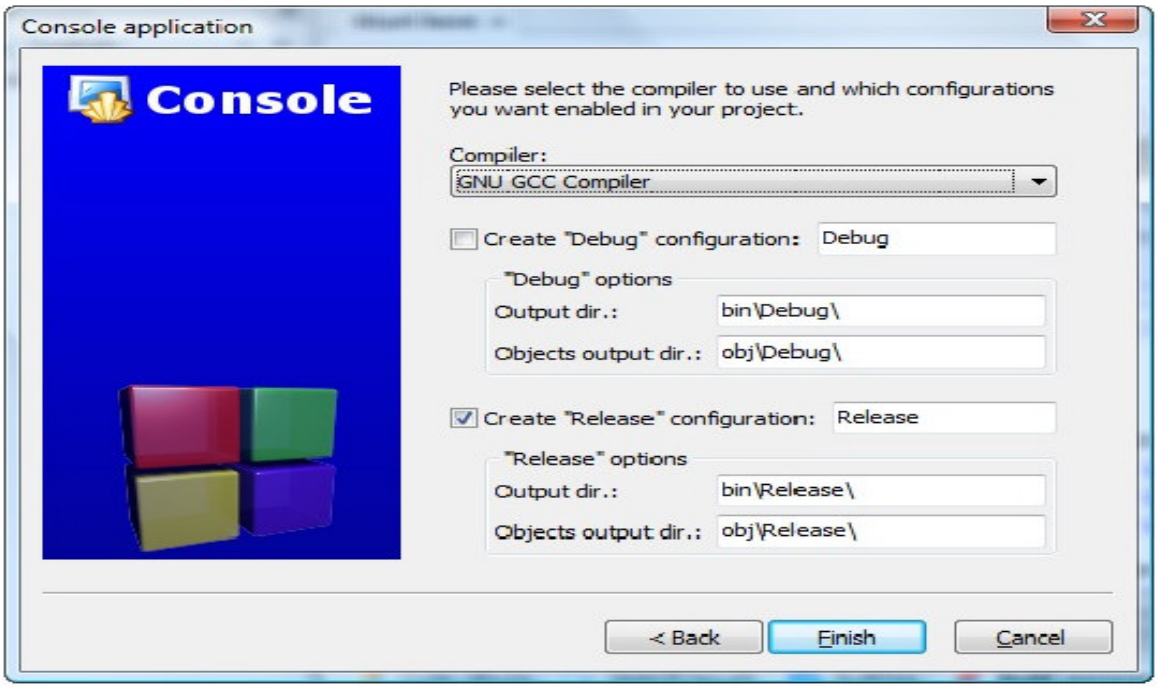
Cliquez sur **"Go"** pour créer le projet. Un assistant s'ouvre.

Faites **"Next",** la première page ne servant à rien. On vous demande ensuite si vous allez faire du C ou du C++ : répondez C++.

On vous demande le nom de votre projet, et dans quel dossier les fichiers source seront enregistrés :



Enfin, la dernière page vous permet de choisir de quelle façon le programme doit être compilé. Vous pouvez laisser les options par défaut, ça n'aura pas d'incidence pour ce que nous allons faire dans l'immédiat (veillez à ce que "Debug" ou "Release" au moins soit coché).



Cliquez sur "Finish", c'est bon !**Code::Blocks** vous créera un premier projet avec déjà un tout petit peu de code source dedans. Dans le cadre de gauche "Project s", développez l'arborescence en cliquant sur le petit "+" pour afficher la liste des fichiers du projet. Vous devriez avoir au moins un main.cpp que vous pourrez ouvrir en double-cliquant dessus.

4-PREMIER PROGRAMME EN LANGAGE C++

Pour vous donner une idée, voici un programme très simple affichant le message "Hello

Code : C++

```
#include <iostream>

using namespace std;

int main()
{
 cout << "Hello world!" << endl;
 return 0;
}
```

world!" à l'écran. Ce sera l'un des premiers codes source que nous étudierons dans les chapitres suivants.

Travaux Pratiques TP N°2

Syntaxe élémentaire, déclaration des variables et opérateurs

2-1-Introduction

Dans ce TP, le but est d'introduire la Syntaxe élémentaire, la déclaration des variables et les opérateurs.

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

➢ Rappel théorique

○ Eléments D'un Programme C++

2-1-1- LES COMMENTAIRES :

Il existe deux types de commentaires en C++ :

- Le premier est symbolisé par une barre oblique et étoile et une autre barre oblique et étoile (/*…….*/), ces commentaires peuvent s'étendre sur plusieurs lignes.
- Le deuxième est symbolisé par deux barres obliques (//…..) qui sont des commentaires d'une seule ligne.

Exemple :

1- /* ces commentaires peuvent s'étendre sur plusieurs lignes.*/

/* indique le début du commentaire et

*/indique la fin du commentaire ou le commentaire compris entre ces deux symboles peut tenir sur plusieurs lignes.

2- // Commentaires d'une seule ligne.

2-2-LES VARIABLES

La liste des types de variables que l'on peut utiliser en C++.

Nom du type	Ce qu'il peut contenir
`bool`	Peut contenir deux valeurs "vrai" (`true`) ou "faux" (`false`).
`char`	Une lettre.
`int`	Un nombre entier.
`unsigned int`	Un nombre entier positif ou nul.
`double`	Un nombre à virgule.
`string`	Une chaîne de caractères. C'est-à-dire une suite de lettres, un mot, une phrase.

Le tableau ci-dessous représente les types de variables avec taille et intervalle :

Type	Size	Range of Values (decimal)
`char`	1 byte	-128 to +127 or 0 to 255
`unsigned char`	1 byte	0 to 255
`signed char`	1 byte	-128 to +127
`int`	2 byte resp. 4 byte	-32768 to +32767 resp. -2147483648 to +2147483647
`unsigned int`	2 byte resp. 4 byte	0 to 65535 resp. 0 to 4294967295
`short`	2 byte	-32768 to +32767
`unsigned short`	2 byte	0 to 65535
`long`	4 byte	-2147483648 to +2147483647
`unsigned long`	4 byte	0 to 4294967295

2-2-1- Déclarer une variable

Il nous faut indiquer à l'ordinateur, le type de la variable que l'on veut, son nom et enfin sa valeur. Pour se faire, c'est très simple.

On indique les choses exactement dans cet ordre.

TYPE NOM (VALEUR) ;

On peut aussi utiliser la même syntaxe que dans le langage C :

TYPE NOM=VALEUR;

Exemple :

```
#include <iostream>

using namespace std;

int main()

{

int age(16);

return 0;

}
```

Que se passe-t-il à la ligne 5 de ce programme ?

L'ordinateur voit que l'on aimerait lui emprunter un tiroir dans sa mémoire avec les propriétés suivantes :

Il peut contenir des nombres entiers.

Il a une étiquette indiquant qu'il s'appelle **âge**.

Il contient la valeur 16

2-2-3 Déclarer sans initialiser

Lors de la déclaration d'une variable, votre programme effectue en réalité deux opérations successives.

1. Il demande à l'ordinateur de lui fournir une zone de stockage dans la mémoire.
2. Il remplit cette case avec la valeur fournie. On parle alors d'**initialisation** de la variable.

Ces deux étapes s'effectuent automatiquement et sans que l'on ait besoin de faire.

Il arrive parfois que l'on ne sache pas quelle valeur donner à une variable lors de sa déclaration. Il est alors possible d'effectuer uniquement l'allocation sans l'initialisation.

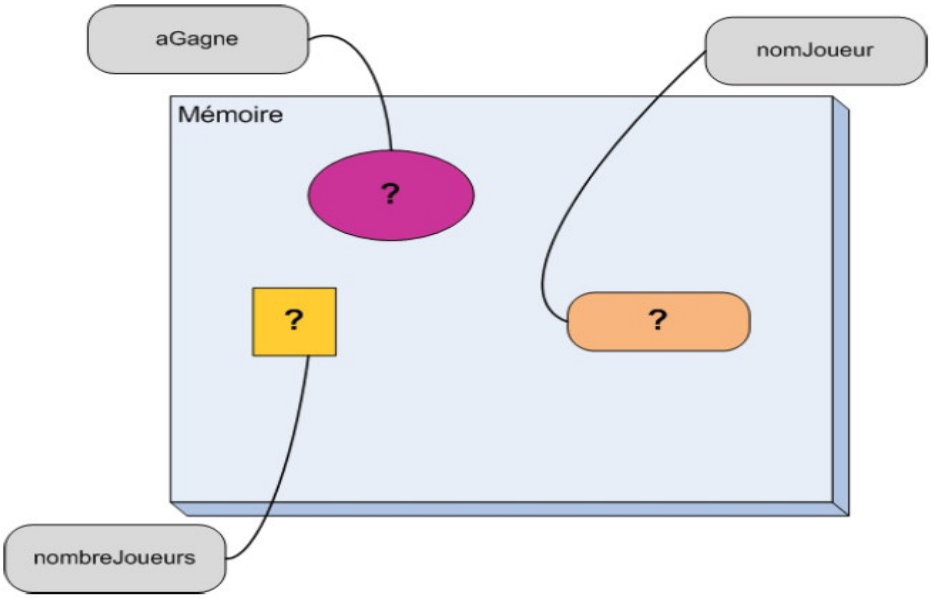
Il suffit d'indiquer le **type** et le **nom** de la variable sans spécifier de valeur.

TYPE NOM ;

Exemple de déclaration sans initialisation

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
string nomJoueur;
int nombreJoueurs;
bool aGagne; //Le joueur a-t-il gagné ?
return 0;

}
```



2-2-3 Afficher la valeur d'une variable

A la place du texte à afficher, on met simplement le nom de la variable.

Code : C++ - Afficher le contenu d'une variable.

cout << âge;

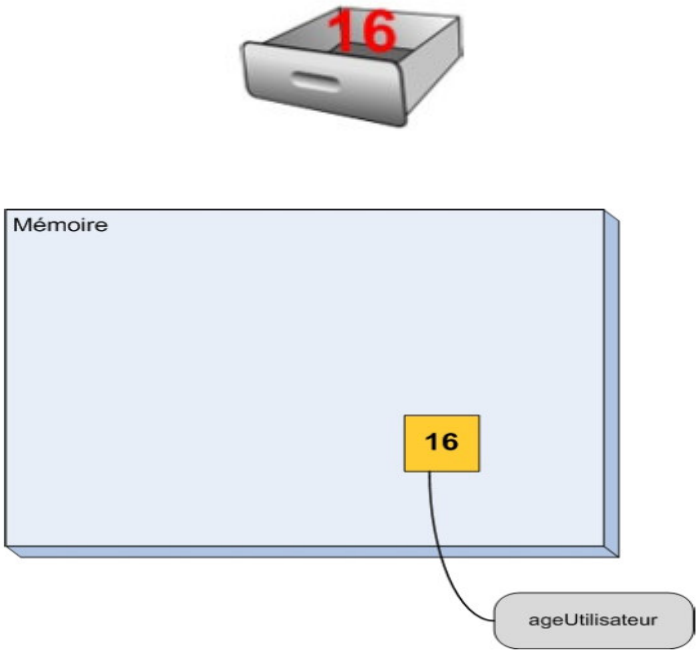
- Exemple d'affichage

```
#include <iostream>
using namespace std;
int main()
{
int age(16);
cout << "Votre âge est : ";
cout << ageUtilisateur;
return 0;
}
```

Une fois compilé, ce code affiche ceci à l'écran :

Code : Console - Résultat du code précédent

```
Votre âge est : 16
```



2-3- LES DIFFERENTS TYPES DES VARIABLES

Les entiers

Le langage C++ distingue plusieurs types entiers

type	description	Taille mémoire
int	Entier standard signé	$4octets: -2^{31} \leq n \leq 2^{31}-1$
Unsigned int	entier positif	$4octets: 0 \leq n \leq 2^{32}$
short	Entier court signé	$2octets: -2^{15} \leq n \leq 2^{15}-1$
Unsigned short	Entier court non signé	$2octets: 0 \leq n \leq 2^{16}$
char	Caractère signé	$1octets: -2^{7} \leq n \leq 2^{7}-1$
Unsigned char	Caractère non signé	$1octets: 0 \leq n \leq 2^{8}$

Quelques constants caractères :

Caractère	
'\n'	Interligne
'\t'	Tabulation horizontale
'\v'	Tabulation verticale
'\r'	Retour chariot
'\f'	Saut de page
'\\'	backslash
'\''	cote
'\'''	guillemets

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	4octets
Double	Réel double précision	8octets

Types de base et constantes

En C++, les types de base sont :

– **bool :** booléen 2, peut valoir true ou false,

– **char :** caractère (en général 8 bits), qui peut aussi être déclaré explicitement signé (signed char) ou non signé (unsigned char),

– **int :** entier (16 ou 32 bits, suivant les machines), qui possède les variantes short [int] et long [int], tous trois pouvant également être déclarés non signés (unsigned),

– **float :** réel (1 mot machine),

– **double:** réel en double précision (2 mots machines), et sa variante long double (3 ou 4 mots machine),

2-4-Les instructions d'affectation et expression

L'affectation simple

```
= affectation
```

Il faut bien noter que le signe = est l'opérateur d'affectation, et non de comparaison ; cela prête parfois à confusion, et entraîne des erreurs difficiles à discerner. À noter aussi que l'affectation est une expression comme une autre, c'est-à-dire qu'elle retourne une valeur. Il est donc possible d'écrire :

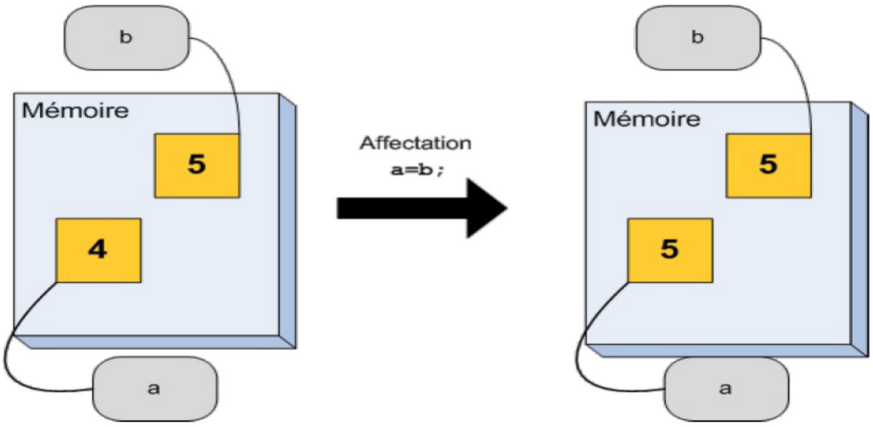
a = b = c+2;

ceci revenant à affecter à b le résultat de l'évaluation de c+2, puis à a le résultat de l'affectation b = c+2, c'est-à-dire la valeur qu'on a donnée à b. Remarquez l'ordre d'évaluation de la droite vers la gauche.

Exemple :Test de l'affectation d'une variable à une autre

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

```
int main()
{
int a(4), b(5); //Déclaration de deux variables.
cout << "a vaut : " << a << " et b vaut : " << b << endl;
cout << "Affectation !"<< endl;
a = b; //Affectation de la valeur de 'b' à 'a'.
cout << "a vaut : " << a << " et b vaut : " << b << endl;
return 0;
}
```



Code : Console(Exécution)

```
a vaut : 4 et b vaut : 5
Affectation !
a vaut : 5 et b vaut : 5
```

L'affectation composée qui consiste à associer un opérateur au signe d'affectation de base '=' :

Forme générale :

Variable opérateur =expression

L'expression est équivalente à :

Variable=variable opérateur expression

Opérateur	Exemple	
	Expression	Equivalent
+=	$a+=8$	$a=a+8$
-=	$a-=8$	$a-=a-8$
$\star=$	$a\star=b$	$a=a\star b$
$/=$	$a/=b+c$	$a/=a/(b+c)$
$\%=$	$a\%=b\star c$	$a=a\%(b\star c)$

3- Opérateurs d'incrémentation et de décrémentation

++ Incrémentation
-- décrémentation

Ces opérateurs, qui ne peuvent être appliqués que sur les types scalaires, peuvent s'employer de deux manières : en principe, s'ils préfixent une variable, celle-ci sera incrémentée (ou décrémentée) avant utilisation dans le reste de l'expression ; s'ils la post fixent, elle ne sera modifiée qu'après utilisation.

Exemple 1 :

A++ ; // A est égale à A+1

A++ ; A=A+1 ; ou A+=1 sont équivalentes.

A-- ; A=A-1 ; ou A-=1 sont équivalentes.

3-5-LES OPERATEURS EN C++

En programmation c++, un opérateur est un symbole qui ordonne au compilateur d'effectuer une opération.

Il existe plusieurs types d'opération :

Opérateur d'opération :

X=a+b ;

Les opérateurs mathématiques :

Il existe cinq opérations l'addition (+), soustraction (-), la multiplication (*), la division (/), et le modulo (%).Un petit tableau récapitulatif Le tableau ci-dessous résume les opérateurs mathématiques.

Opération	Symbole	Exemple
Addition	+	**résultat = a + b;**
Soustraction	-	**résultat = a - b;**
Multiplication	*	**résultat = a * b;**
La division	/	**résultat = a / b;**
modulo	%	résultat = a %b;

Exemple :

```
#include <iostream>
using namespace std;
int main()
{
    int a = 10, b = 20, c, d, e;
    c = a + b;          //c = 10+20=30
    d = a * c;          // d=10*30=300
    d = d - 80;         // d=300-80= 220
    e = d / 7;          // e=220/7=31
    cout << e % 4 << endl;          // reste (31/4)
    return 0; }
```

Code : Console (exécution)

```
3
```

Nous terminons en donnant quelques fonctions de la bibliothèque standard.

Fonctions mathématiques : elles sont déclarées dans **<cmath>.**il y a notamment les fonctions suivantes, de paramètre double et de résultat double :

4- **Floor :** (resp.ceil) : partie entière par défaut
5- **Fabs** : valeur absolue
6- **Sqrt** :racine carrée
7- **Pow** : puissance (pow(x,y) renvoie x^y)
8- **Exp, log, log10**
9- **Sin, cos, tan, asin, acos, atang, sinh, cosh, tanh**
10- **Les opérateurs relationnels**

Il en existe six (06) :
Égale à ⟶ ==,
Différent de ⟶ (!=)
Supérieur à ⟶ (>)
Supérieur ou égale à ⟶ (>=)
Inférieur à ⟶ (<)
Inférieur ou égale à ⟶ (<=)

Symbole	Signification
==	Est égal à
>	Est supérieur à
<	Est inférieur à
>=	Est supérieur ou égal à
<=	Est inférieur ou égal à
!=	Est différent de

Exemple :

```
Int a=27 ; b=19 ;
a==b ; //cette opération donne la valeur False
a<b ; //cette opération donne la valeur True
```

Les opérateurs d'entrée/sortie

L'opération d'entrée est symbolisée par >> <variable> : lecture au clavier de la valeur de <variable> et l'opération de sortie est symbolisée par : << <expression>, affichage à l'écran de la valeur de <expression>.

Exemple :

```
Int a ;
Cin >>//pour entrer une valeur pour A
A++
Cout <<A ;// afficher le résultat.
```

Voici un tableau présentant l'ensemble des opérateurs de C++ (certains ne seront exploités que dans des chapitres ultérieurs) :

Catégorie	Opérateurs
Résolution de portée	:: (portée globale - unaire) :: (portée de classe - binaire)
Référence	() [] -> .
Unaire	+ - ++ -- ! ~ * & sizeof cast dynamic_cast static_cast reinterpret_cast const_cast new new[] delete delete[]
Sélection	->* .*
Arithmétique	* / %
Arithmétique	+ -
Décalage	<< >>
Relationnels	< <= > >=
Relationnels	== !=
Manipulation de bits	&
Manipulation de bits	^
Manipulation de bits	|
Logique	&&
Logique	||
Conditionnel (ternaire)	? :
Affectation	= += -= *= /= %= &= ^= |= <<= >>=
Séquentiel	,

Pour avoir accès à plus de fonctions mathématiques, il faut ajouter une ligne en haut de votre programme, comme lors que l'on désire utiliser des variables de type string. Il faut ajouter

Code : C++ - Ligne à ajouter pour les fonctions mathématiques

```
#include <cmath>
```

Quelques autres fonctions présentes dans **cmath** Comme il y a beaucoup de fonctions, je vous propose de tout mettre dans un tableau.

Nom de la fonction	Symbole mathématique	Nom de la fonction en C++	Mini-exemple
Racine carrée	$\sqrt{x}$	`sqrt()`	`resultat = sqrt(valeur);`
Sinus	$\sin(x)$	`sin()`	`resultat = sin(valeur);`
Cosinus	$\cos(x)$	`cos()`	`resultat = cos(valeur);`
Tangente	$\tan(x)$	`tan()`	`resultat = tan(valeur);`
Exponentielle	e^x	`exp()`	`resultat = exp(valeur);`
Logarithme népérien	$\ln x$	`log()`	`resultat = log(valeur);`
Logarithme en base 10	$\log_{10} x$	`log10()`	`resultat = log10(valeur);`
Valeur absolue	$\lvert x \rvert$	`fabs()`	`resultat = fabs(valeur);`
Arrondi vers le bas	$\lfloor x \rfloor$	`floor()`	`resultat = floor(valeur);`
Arrondi vers le haut	$\lceil x \rceil$	`ceil()`	`resultat = ceil(valeur);`

2-PARTIE PRATIQUE :

Problème :

1-Calculer l'addition et le produit de deux entiers ?

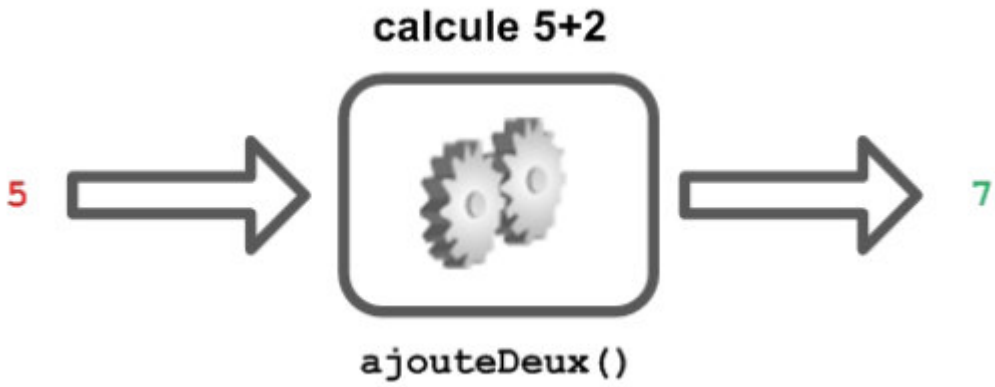
2-Calculer La racine carrée ?

SOLUTION

1-1-L'addition de deux nombres :

```
#include <iostream>
using namespace std;
int main()
{
double a(0), b(0); //Déclaration des variables utiles
cout << "Bienvenue dans le programme d'addition a+b !" << endl;
cout << "Donnez une valeur pour a : "; //Demande du premier
nombre
cin >> a;
cout << "Donnez une valeur pour b : "; //Demande du deuxième
nombre
cin >> b;
```

```
double const resultat(a + b); //On effectue l'opération
cout << a << " + " << b << " = " << resultat << endl; //On
affiche le résultat
return 0;
   }
```



Code : Console (exécution)

```
Bienvenue dans le programme d'addition a+b !
Donnez une valeur pour a : 123.784
Donnez une valeur pour b : 51.765
   123.784 + 51.765 = 175.549
```

1-2-Le produit de nombres

Soit deux entiers a et b données, on veut calculer leur produit p=a*b

Algorithme :

A,b :entiers

P :entier

Debut

Ecrire('introduire l'entier a')

Lire(a)

Ecrire('introduire l'entier b')

Lire(b)

p ← a*b

Ecrire('introduire l'entier p')

Fin

Programme en c++ :

```
#include <iostream>
using namespace std;
int main(
)
{
int a,b,p;
/*a,b,p trois entiers*/
/*ce programme calcule le produit de deux nombres entiers*/
cout << "Donnez un nombre entier a: \n";
cin>>a;
/*lecture de a*/
cout<<"Donnez un nombre entier b: \n";
//Saut de lingne
cin>>b;
/*lecture de b*/
p=a*b;
/*calcul du produit de a et b*/
cout<<"\n";
//Saut de lingne
cout<<"le produit de a et b est :"<<p<<"\n";
return 0;
}
```

2- Prenons un exemple complet, je pense que vous allez comprendre rapidement le principe.

```
#include <iostream>
#include <cmath> //Ne pas oublier cette ligne
using namespace std;
```

```
int main()
{
    double const nombre(16);        //Le nombre dont on veut la
racine
                                    //Comme sa valeur ne
changera
pas on met 'const'
    double resultat;                //Une case mémoire pour
stocker le résultat
    resultat = sqrt(nombre);  //On effectue le calcul !
    cout << "La racine de " << nombre << " est " << resultat
<<
endl;
    return 0;
  }
```

Code : Cons ole

```
La racine de 16 est 4
```

3-Travail demandé :

Calcule périmètre et la surface d'un cercle

Ecrire le programme qui calcule périmètre et la surface d'un cercle

SOLUTION DU TRAVAIL DEMANDE

Programme en C++

```
#include <iostream>
using namespace std;
int main()
{
float pi=3.1415;
float rayon,perim,surf;
cout<<"saisair le rayon du cercle \n";
cin>>rayon;
perim=2*rayon*pi;
surf=rayon*rayon*pi;
cout<<"le perimetre du cercle est "<<perim<<"\n";
cout<<"la surface du cercle est "<<surf<<"\n";
return 0;
}
```

Travaux Pratiques TP N°3

Structures conditionnelles et boucles

- # INTRODUTION

Dans ce TP, le but est de maitriser la programmation des structures conditionnelles et les boucles avec C++.

- ## Rappel théorique

Une structure de contrôle ou instruction de contrôle sert à contrôler le déroulement d'un traitement.

Un traitement peut s'exécuter de différentes manières :

11- Séquentiellement (l'un à la suite de l'autre).

12- Alternativement (soit l'un soit l'autre ou les autres selon une condition fixée).

Alternative simple (if), alternative composée (if else), alternative imbriquée (if…if else else), alternative multiple (switch)

13- Répétitivement (en répétant le traitement un nombre fini de fois).

La boucle while, la boucle do while, la boucle for

Le terme instruction désignera indifféremment : une instruction simple (terminée par un point-virgule), une instruction structurée (choix, boucle) ou un bloc (instructions entre {et}).

Structures de contrôle.

- ## LES CONDITIONS

Pour effectuer ces tests de structure conditionnelle, nous utilisons des symboles. Voici le tableau des symboles à connaître par cœur :

Symbole	Signification
==	Est égal à
>	Est supérieur à
<	Est inférieur à
>=	Est supérieur ou égal à
<=	Est inférieur ou égal à
!=	Est différent de

Il faut savoir qu'il existe plusieurs types de conditions en C++ pour faire des tests, mais la plus importante qu'il faut impérativement connaître est sans aucun doute la condition if.

o La condition if

Les conditions permettent de tester des variables pour qu'un programme soit capable de prendre des décisions.

- **Syntaxe des traitements alternatifs if**

L'alternative simple :	**if (condition)** **Séquence**
L'alternative composée :	**if (condition)** **Séquence 1** **else** **Séquence 2 ;**
L'alternative imbriquée :	**if (condition1)** **{** **if (condition2)** **Séquence 1 ;** **else** **{** **Séquence 3 ;** **}** **}** **else** **Séquence 4 ;**

- **Syntaxe des traitements alternatifs switch**

L'alternative switch :	**switch(séquence)** **{** **Case valeur 1 :séquence 1 ;**

```
                                    Break ;
.
                                    Case valeur 2 :séquence 2 ;
                                    Case valeur N :séquence N ;
                                    Break ;
                                    Default :séquence ;
                                    }
```

- **Exemple l'alternative switch**

En théorie, la condition **if** permet de faire tous les tests que l'on veut. En pratique, il existe d'autres façons de faire des tests. La condition switch, par exemple, permet de simplifier l'écriture de conditions qui testent plusieurs valeurs différentes pour une même variable.

Prenez par exemple le test qu'on vient de faire sur le nombre d'enfants :

A-t-il 0 enfants ?

A-t-il 1 enfant ?

A-t-il 2 enfants ?

...

On peut faire ce genre de tests avec des **if... else if... else**, mais on peut faire la même chose avec une condition **switch** qui a tendance à rendre le code plus lisible dans ce genre de cas. Voici ce que donnerait la condition précédente avec un **switch** :

Code : C++

```
#include <iostream>
using namespace std;
int main()
{
   int nbEnfants(2);
switch (nbEnfants)
{
case 0:
cout << "Eh bien alors, vous n'avez pas d'enfants ?" <<
```

```
endl;
break;
case 1:
cout << "Alors, c'est pour quand le deuxieme ?" << endl;
break;
case 2:
cout << "Quels beaux enfants vous avez la !" << endl;
break;
default:
cout << "Bon, il faut arreter de faire des gosses
maintenant !" << endl;
break;
}
return 0;
   }
```

Cela affiche :

Code : Console

```
Quels beaux enfants vous avez la !
```

La forme est un peu différente : on indique d'abord qu'on va analyser la variable nbEnfants(ligne 9). Ensuite, on teste tous les cas (**case**) possibles : si ça vaut 0, si ça vaut 1, si ça vaut 2...

Les **break** sont obligatoires si on veut que l'ordinateur ne continue pas d'autres tests une fois qu'il en a vérifié un.

Enfin, le **default** à la fin correspond au **else** ("sinon") et s'exécute si aucun test précédent n'est vérifié.

o LES BOUCLES

Le principe des boucles 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 :

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

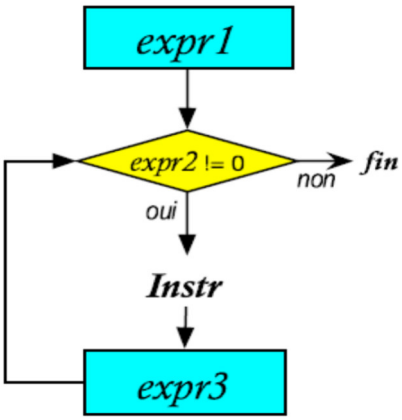
- **La boucle FOR**

Ce type de boucle, que l'on retrouve fréquemment, permet de condenser :

- Une initialisation
- Une condition
- Une incrémentation

Syntaxe :

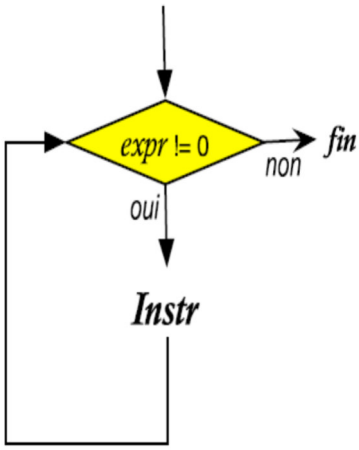
for (expr1 ;expr2 ;expr3) instr



- **La boucle while**

Syntaxe:

```
while (condition)
{
/* Instructions à répéter */
}
```



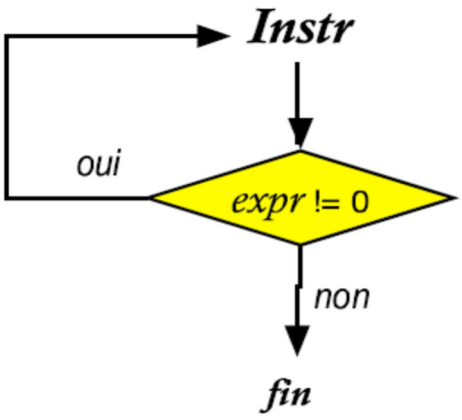
Tout ce qui est entre accolades sera répété tant que la condition est vérifiée.

- **La boucle do while**

Syntaxe:

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**.

```
do
{
/* Instructions */
} while (condition);
```



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

- **Exemple :**

Soit une équation du second degré $ax^2+bx+c=0$

a,b,c sont des nombres réels.

Ecrire un programme qui permet de résoudre cette équation.

Pour trouver les solutions si elles existent il faut :

1-verifier si a=0 dans ce cas la solution est x=-c/b

2-calculer le discriminant $D=b^2-4ac$

Si D=0 alors il existe une solution double x0=-b/2a

Si D>0 alors il existe deux solutions $x1=(-b-\sqrt{D})/2a$

$X2=(-b+\sqrt{D})/2a$

Si D<0 alors pas de solutions réelles

Code c++ :

```
#include <iostream>
#include <math.h>// pour l'utilisation de la fonction sqrt
using namespace std;
```

```
int main()
{
float a,b,c,x0,x1,x2,D;
cout<<"resolution d'une equation du second degré \n";
cout<<"introduire la valeur de a \n";
cin>>a;
cout<<"introduire la valeur de b \n";
cin>>b;
cout<<"introduire la valeur de c \n";
cin>>c;
if (a==0 && b!=0)
{
Cout<< "on obtient une équation du premier degré \n" ;
X0=-c/b ;
Cout<< "la solution de l'équation est : "<<x0<<"  \n" ;
}
Else
{
D=(b*b)-4*a*c ;
If (D==0)
{
X0=-b/(2*a);
Cout<< "la solution  double de l'équation est : "<<x0<<"  \n" ;
}
Else
If (D>0)
{
X1=(-b-sqrt(D))/(2*a) ;
X2=(-b+sqrt(D))/(2*a) ;
Cout<< "deux solutions distinctes de l'équation \n" ;
Cout<< "x1= "<<x1<<"  \n" ;
```

```
Cout<< "x2= "<<x2<<"  \n" ;
}
Else
     {
Cout<< "pas de solutions réelles  \n" ;
    }
}
return 0;
}
```

- **Problème :**

Calcul du plus grand commun diviseur (PGCD)

Soit x et y deux nombres positifs, non nuls tels que x>y.

Le programme qui permet de calculer le plus grand commun diviseur (PGCD).

Analyse/indications

Le PGCD (plus grand commun diviseur), est le nombre qui divise x et y en même temps il est supérieur à tous les autres diviseurs de x et y.

Exemple :

X=20,y=15

Diviseurs de x={1,2,4,5,10,,20]

Diviseurs de x={1,2,5,15]

Diviseurs communs={1,5]

PGCD=5

Programme en c++ :

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

int main()
{
   int a,b,pgcd;
```

```
    cout << "donner a:" ;
    cin>>b;
    cout << "donner b:" ;
    cin>>b;
    cout << "pgcd("<<a<<","<<b<<")=" ;
    do
    {
       if (a>b)
       {
          a=a-b;
       }
       else
       {
          b=b-a;
       }
    }while (a!=b);
    pgcd=a;
    cout<<pgcd;
    return 0;
}
```

- **Travail demandé :**

Calcule du PPCM de deux nombres

Ecrire le programme qui calcule le plus petit commun multiple PPCM de deux nombres

Solution du travail demandé

```
#include <iostream>
using namespace std;
int main()
{
  int a,b,pgcd,ppcm,n;
   cout << "donner a:" ;
   cin>>a;
   cout << "donner b:" ;
   cin>>b;
   n=a*b;
   cout << "pgcd("<<a<<","<<b<<")=" ;
   do
   {
      if (a>b)
      {
         a=a-b;
      }
      else
      {
         b=b-a;
      }
   }while (a!=b);
   pgcd=a;
   cout<<pgcd<<endl;
   ppcm=n/pgcd;
   cout<<"le ppcm entre a et best:"<<ppcm;
   return 0;
}
```

Travaux Pratiques TP N°4

Les tableaux et les pointeurs

✓ Introduction

Dans ce TP, le but est d'introduire les tableaux et les pointeurs.

✓ Rappel théorique

o TABLEAUX

Ils existent deux sortes de tableaux différents. Ceux dont la taille est fixée **les tableaux statiques** et ceux dont la taille peut varier, les **tableaux dynamiques**.

- **Forme de la déclaration d'un tableau statique**

<type> <nom> [<taille>];

où :

<type> est le type des ´éléments du tableau,

<nom> est le nom du tableau,

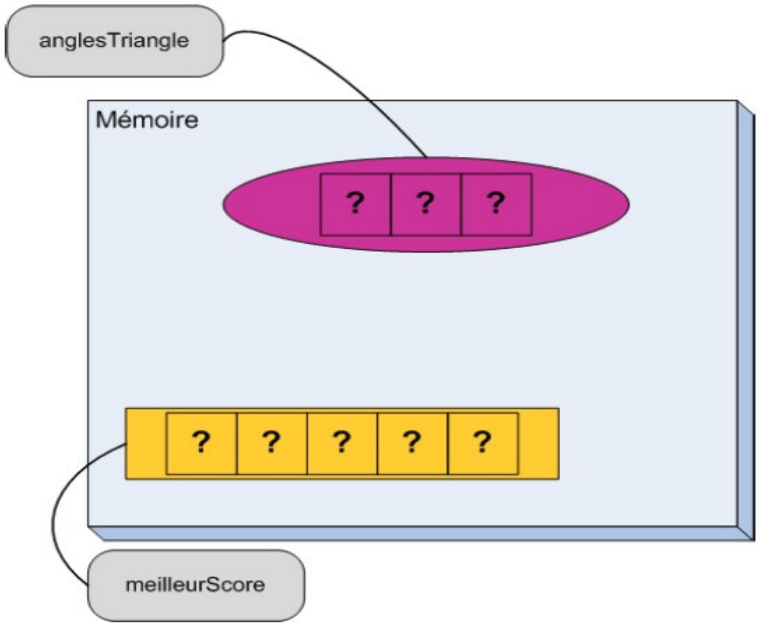
<taille> est une constante entière ´égale au nombre d'éléments du tableau.

On indique le type, puis le nom choisi et enfin la taille du tableau entre crochets. Voyons ça avec un exemple.

Exemple:

Code : C++: tableau

```
#include <iostream>
using namespace std;
int main()
{
int meilleurScore[5]; //Declare un tableau de 5 entiers
double anglesTriangle[3]; //Declare un tableau de 3 double
return 0;
}
```



On retrouve nos deux zones mémoires avec leurs étiquettes, mais cette fois, chaque zone est découpée en cases. Trois cases pour le tableau **anglesTriangle** et cinq cases pour le tableau **meilleurScore**. Pour l'instant toutes ces cases ne sont pas initialisées. Leur contenu est donc quelconque.

- **Accéder aux éléments d'un tableau**

Pour accéder à une case on utilise la syntaxe :

nomDuTableau[numeroDeLaCase]

Il y a juste une petite subtilité, la première case possède le numéro 0 et pas 1. Tout est en quelque sorte décalé de 1. Pour accéder à la 3e case de **meilleurScore** et y écrire une valeur, il faudra donc écrire :

Code : C++

```
meilleurScore[2] = 5;
```

En effet,**3-1=2** , la **3**e case possède le numéro **2**. Si je veux remplir mon tableau des meilleurs scores comme dans l'exemple initial, je peux donc écrire :

Code : C++ - Remplissage d'un tableau

```
int const nombreMeilleursScores(5); //La taille du tableau
int meilleursScores[nombreMeilleursScores]; //Declaration du
tableau
meilleursScores[0] = 118218; //Remplissage de la premiere case
meilleursScores[1] = 100432; //Remplissage de la deuxiemecase
meilleursScores[2] = 87347; //Remplissage de la troisieme case
meilleursScores[3] = 64523; //Remplissage de la quatrieme case
meilleursScores[4] = 31415; //Remplissage de la cinquieme case
```

- **Parcourir un tableau**

Le gros point fort des tableaux, c'est qu'on peut les parcourir en utilisant une boucle. On peut ainsi effectuer une action sur chacune des cases d'un tableau l'une après l'autre.

Code : C++ - Parcourir un tableau

```
int const nombreMeilleursScores(5); //La taille du tableau
int meilleursScores[nombreMeilleursScores]; //Declaration du
tableau
meilleursScores[0] = 118218; //Remplissage de la premiere case
meilleursScores[1] = 100432; //Remplissage de la deuxiemecase
meilleursScores[2] = 87347; //Remplissage de la troisieme case
meilleursScores[3] = 64523; //Remplissage de la quatrieme case
meilleursScores[4] = 31415; //Remplissage de la cinquieme case
for(int i(0); i<nombreMeilleursScores; ++i)
{
cout << meilleursScores[i] << endl;
}
```

La variable **i** va prendre successivement les valeurs **0,1,2,3** et **4**. Ce qui veut dire que ce seront les valeurs de **meilleursScores[0]** puis **meilleursScores[1]** etc. qui seront envoyées dans **cout**.

Exemple:

Voici un programme qui permet d'entrer des entiers dans un vecteur de 6 elements et de calculer la somme de ses elements ?

Code C++

```
#include <iostream>
using namespace std;
int main()
{
Int tab[6];
Int i;
Int som=0;
For(i=1;i<=5;i++)
{
cout<<"introduire un entier dans le tableau,tab["<<i<<"]=";
cin>>tab[i];
som=som+tab[i];
}
cout<<"la somme des elements du tableau est "<<som<<"\n";
return 0;
}
```

- **Déclarer un tableau dynamique**

La première différence se situe au tout début de votre programme. Il faut ajouter la ligne **#include <vector>** pour utiliser ces tableaux.

On utilise la syntaxe :

vector<type> nom(VectTaille);

Par exemple pour un tableau de 5 entiers, on écrit :

Code : C++

```
#include <iostream>
#include <vector> //Ne pas oublier !
using namespace std;
int main()
{
vector<int> tableau(5);
return 0;
}
```

- **Accéder aux éléments d'un tableau**

La déclaration était très différente des tableaux statiques. Par contre là, c'est exactement identique. On utilise à nouveau les crochets et la première case possède aussi le numéro 0.

Exemple :

Code : C++ - Remplissage d'un tableau

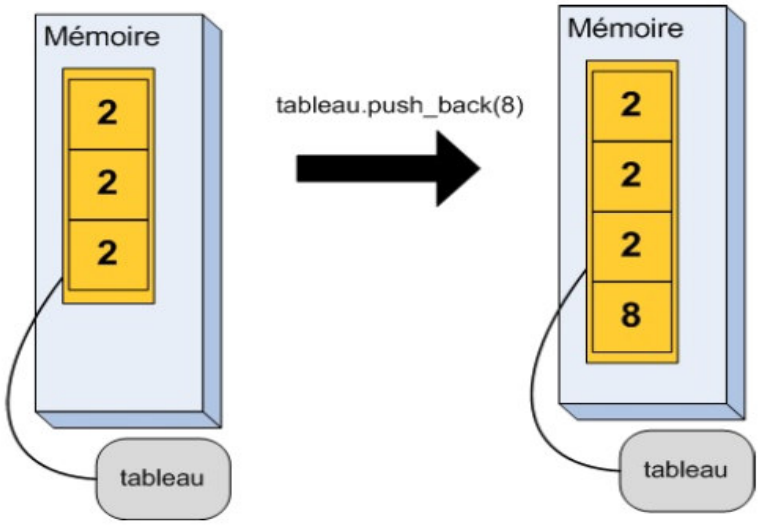
```
int const nombreMeilleursScores(5); //La taille du tableau
vector<int> meilleursScores(nombreMeilleursScores); //Déclaration
du tableau
meilleursScores[0] = 118218; //Remplissage de la première case
meilleursScores[1] = 100432; //Remplissage de la deuxième case
meilleursScores[2] = 87347; //Remplissage de la troisième case
meilleursScores[3] = 64523; //Remplissage de la quatrième case
meilleursScores[4] = 31415; //Remplissage de la cinquième case
```

- **Changer la taille**

Entrons maintenant dans le vif du sujet. Faire varier la taille d'un tableau. Commençons par ajouter des cases à la fin d'un tableau. Il faut utiliser **push_back()**. On écrit le nom du tableau, suivi d'un point et du mot **push_back** avec entre parenthèses la valeur qui va remplir la nouvelle case.

Code : C++

```
vector<int> tableau(3,2); //Un tableau de 3 entiers valant tous 2
tableau.push_back(8); //On ajoute une 4ème case au tableau. Cette case contient la valeur .
```



On peut supprimer la dernière case d'un tableau en utilisant la fonction **pop_back()** de la même manière que **push_back().** Sauf qu'il n'y a rien à mettre entre les parenthèses.

Code : C++

```
vector<int> tableau(3,2); //Un tableau de 3 entiers valant tous 2
tableau.pop_back(); //Et hop ! Plus que 2 cases.
tableau.pop_back(); //Et hop ! Plus qu'une case.
```

Il y a une fonction pour ça. C'est la fonction **size().** En faisant **tableau.size(),** on récupère un entier correspondant au nombre d'éléments de tableau.

Code : C++

```
vector<int> tableau(5,4); //Un tableau de 5 entiers valant tous 4
int const taille(tableau.size()); //Une variable pour contenir la taille du tableau
//La taille peut varier mais la valeur de cette variable ne changera pas.
//On utilise donc une constante.
//A partir d'ici, la constante 'taille' vaut donc 5
```

Exemple : du calcul des moyennes

Code : C++ - Calcul de moyenne en utilisant vector

```
#include <iostream>
#include <vector> //Ne pas oublier !!
using namespace std;
int main()
{
vector<double> notes; //Un tableau vide
notes.push_back(12.5); //On ajoute des cases avec les notes
notes.push_back(19.5);
notes.push_back(6);
notes.push_back(12);
notes.push_back(14.5);
notes.push_back(15);
double moyenne(0);
for(int i(0); i<notes.size(); ++i) //On utilise notes.size()
pour la limite de notre boucle
{
moyenne += notes[i]; //On additionne toutes les notes
}
moyenne /= notes.size(); //On utilise a nouveau notes.size()
pour obtenir le nombre de notes
cout << "Votre moyenne est : " << moyenne << endl;
return 0;

}
```

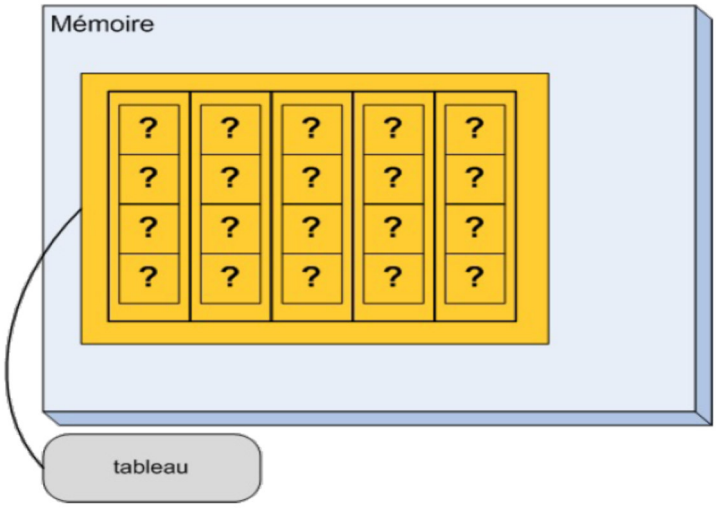
1. Tableau multi-dimensionnel

- **Déclaration d'un tableau multi-dimensionnel**

Pour déclarer un tel tableau, il faut indiquer les dimensions les unes après les autres entre crochets.

type nomTableau[tailleX][tailleY]

Exemple :



Donc pour reproduire le tableau du schéma, on doit déclarer le tableau suivant.

```
int tableau[5][4];
```

Ou encore mieux, en déclarant des constantes.

Code : C++ - Déclaration du tableau du schéma

```
int const tailleX(5);

int const tailleY(4);
int tableau[tailleX][tailleY];
```

- **Accéder aux éléments**

Pour accéder à une case de notre grille, il faut indiquer la position en X et en Y de la case voulue. Par exemple **tableau[0][0]** accède à la case en-bas à gauche de la grille. **tableau[0][1]** correspond à celle qui se trouve juste en-dessus, alors que **tableau[1][0]** se situe directement à sa droite.

1. 2-Les pointeurs

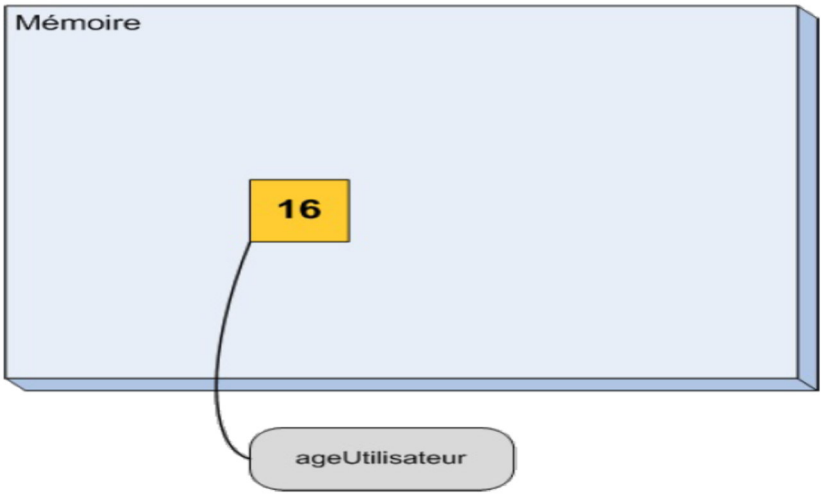
1. Adresse

Lorsque l'on déclare une variable, l'ordinateur nous prête une place dans sa mémoire et y accroche une étiquette portant le nom de la variable.

Code : C++

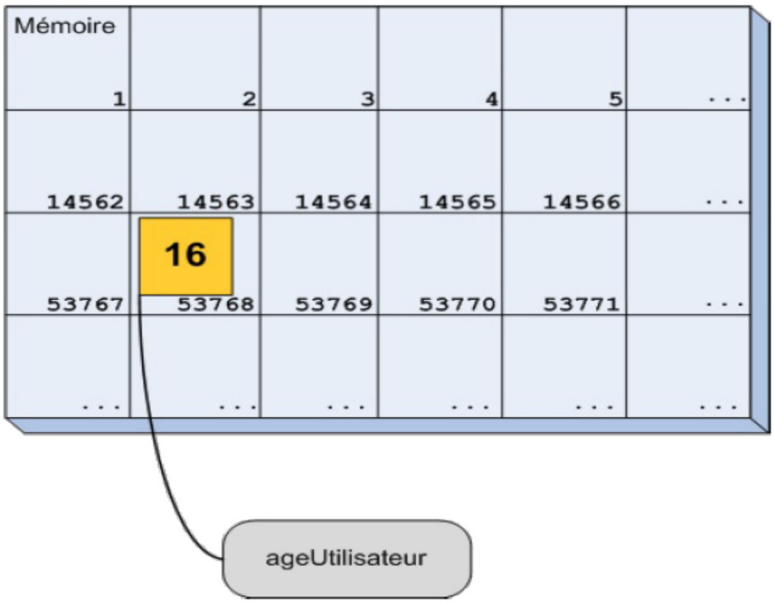
```
int main()
{
int ageUtilisateur(16);
return 0;
}
```

On pouvait donc représenter la mémoire utilisée dans ce programme sur le schéma suivant :



La mémoire d'un ordinateur est réellement constituée de "cases". Il y en a même énormément. Plusieurs milliards sur un ordinateur récent ! Il faut donc un système pour que l'ordinateur puisse retrouver les cases dont il a besoin.

Chaque "case" possède un numéro unique, son **adresse**.



L'important est que chaque variable possède une seule adresse. Et chaque adresse correspond à une seule variable. L'adresse est donc un deuxième moyen d'accéder à une variable. On peut accéder à la case jaune du schéma par deux chemins différents :

1. On peut passer par **son nom** (l'étiquette) comme on sait déjà le faire...
2. Mais on peut aussi accéder à la variable grâce à son **adresse** (son numéro de case).. On pourrait alors dire à l'ordinateur "*Affiche moi le contenu de l'adresse 53768*"

- **Afficher l'adresse**

En C++, le symbole pour obtenir l'adresse d'une variable est l'espérluette **(&)**. Si je veux afficher l'adresse de la variable **ageUtilisateur**, je dois donc écrire **&ageUtilisateur**. Essayons.

Code : C++

```
#include <iostream>
using namespace std;
int main()
{
int ageUtilisateur(16);
cout << "L'adresse est : " << &ageUtilisateur << endl; //
Affichage de l'adresse de la variable
return 0;
```

```
}
```

Code : Console

```
L'adresse est : 0x22ff1c
```

L'esperluette veut dire "adresse de". Donc cout << &a; se traduit en français par "Affiche l'adresse de la variable a".

1. Les pointeurs définition

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

Une variable pointeur est destiné à manipuler des adresses d'informations d'un type donné.

- **Déclarer un pointeur**

Pour déclarer un pointeur, il faut, comme pour les variables, deux choses :

Un type

Un nom

Pour le nom, il n'y a rien de particulier à signaler. Les mêmes règles que pour les variables s'appliquent. Ouf !

Le type d'un pointeur a une petite subtilité. Il faut indiquer quel est le type de variable dont on veut stocker l'adresse et ajouter une étoile **(*)**.

Exemple :

Code : C++ - Déclaration d'un pointeur

```
int *pointeur;
```

Ce code déclare un pointeur qui peut contenir l'adresse d'une variable de type int.

Exemple :

Code : C++ - Déclaration de pointeurs

```
double *pointeurA; //Un pointeur qui peut contenir l'adresse d'un nombre a virgule
unsigned int *pointeurB; //Un pointeur qui peut contenir l'adresse d'un nombre entier positif
string *pointeurC; //Un pointeur qui peut contenir l'adresse d'une chaîne de caractères
vector<int> *pointeurD; //Un pointeur qui peut contenir l'adresse d'un tableau dynamique de nombres entiers
```

```
int const *pointeurE; //Un pointeur qui peut contenir l'adresse d'un nombre entier constant
```

- **Stocker une adresse**

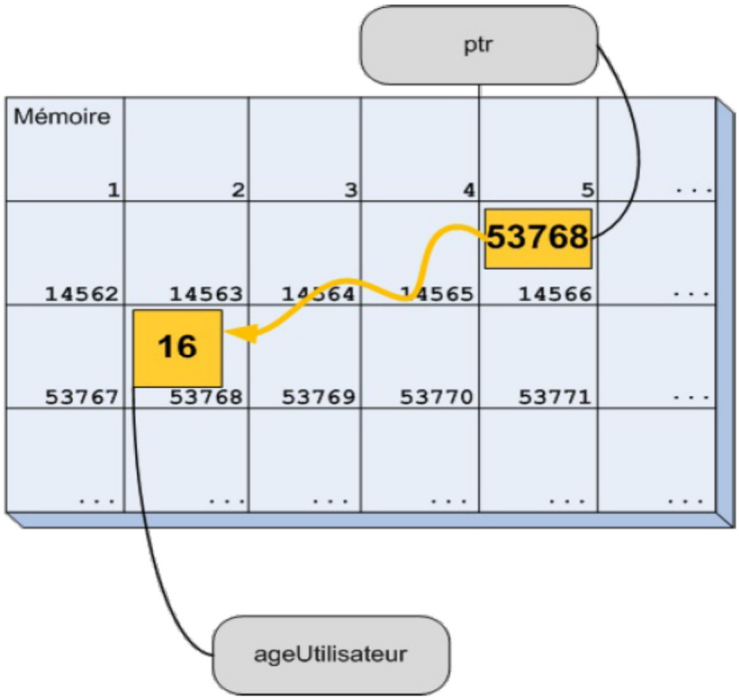
Maintenant qu'on a la variable, il n'y a plus qu'à mettre une valeur dedans. Vous savez déjà comment obtenir l'adresse d'une variable (rappelez-vous du **&**). Allons-y !

Code : C++

```
int main()

{
int ageUtilisateur(16); //Une variable de type int.
int *ptr(0); //Un pointeur pouvant contenir
l'adresse d'un nombre entier.
ptr = &ageUtilisateur; //On met l'adresse de 'ageUtilisateur'
dans le pointeur 'ptr'.
return 0;
}
```

La ligne 6 est celle qui nous intéresse. Elle écrit l'adresse de la variable ageUtilisateur dans le pointeur ptr. On dit alors que le pointeur **ptr pointe** sur ageUtilisateur.



- **Afficher l'adresse**

Comme pour toutes les variables, on peut afficher le contenu d'un pointeur.

Code : C++

```
#include <iostream>
using namespace std;
int main()
{
int ageUtilisateur(16);
int *ptr(0);
ptr = &ageUtilisateur;
cout << "L'adresse de 'ageUtilisateur' est : " <<
&ageUtilisateur << endl;
cout << "La valeur de pointeur est : " << ptr << endl;
return 0;
}
```

Code : Console

```
L'adresse de 'ageUtilisateur' est : 0x2ff18
La valeur de pointeur est : 0x2ff18
```

- **Accéder à la valeur pointée**

Le but des pointeurs est d'accéder à une variable sans passer par son nom. Voici comment faire. Il faut utiliser l'étoile **(*)** sur le pointeur pour afficher la valeur de la variable pointée.

Code : C++

```
int main()
{
int ageUtilisateur(16);
int *ptr(0);
ptr= &ageUtilisateur;
cout << "La valeur est : " << *ptr << endl;
return 0;

}
```

En faisant **cout** << ***ptr**, le programme va effectuer les étapes suivantes :

1. Aller dans la case mémoire nommée **ptr**.
2. Lire la valeur enregistrée.
3. "Suivre la flèche" pour aller à l'adresse pointée.
4. Lire la valeur stockée dans la case.
5. Afficher cette valeur. Ici, ce sera bien sûr 16 qui sera affiché.

En utilisant l'étoile, on accède à la **valeur de la variable pointée**. C'est ce qui s'appelle **déréférencer** un pointeur.

- **L'allocation dynamique**

Pour demander manuellement une case dans la mémoire, il faut utiliser l'opérateur **new**.

new va demander une case à l'ordinateur et renvoyer un **pointeur** pointant vers cette case.

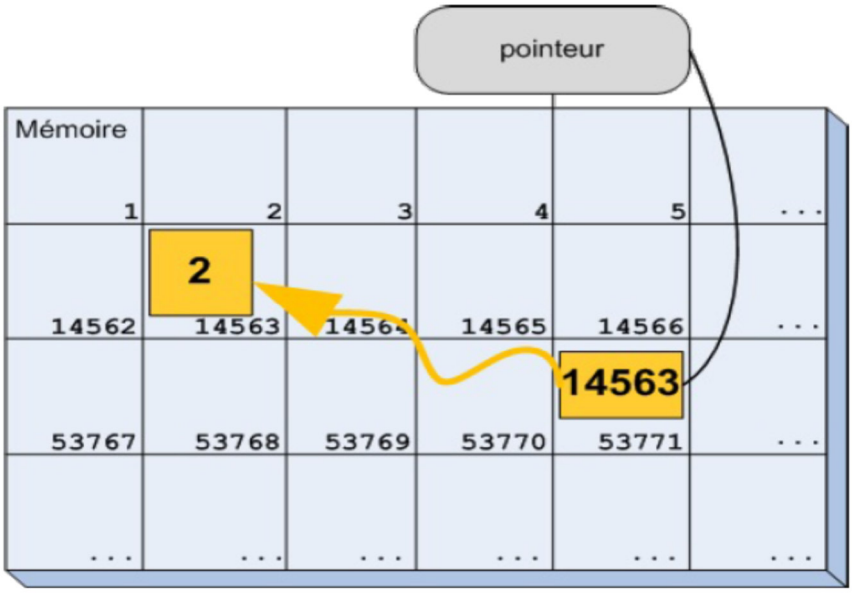
Code : C++

```
int *pointeur(0);
pointeur = new int;
*pointeur = 2; //On accède à la case mémoire pour en modifier la Valeur
```

La deuxième ligne demande une case mémoire pouvant stocker un entier et l'adresse de cette case est stockée dans le pointeur.

Une fois allouée manuellement, la variable s'utilise comme n'importe quelle autre. On doit juste se rappeler qu'il faut y accéder par le pointeur en le déréférençant.

La case sans étiquette est maintenant remplie. La mémoire est donc dans l'état suivant :



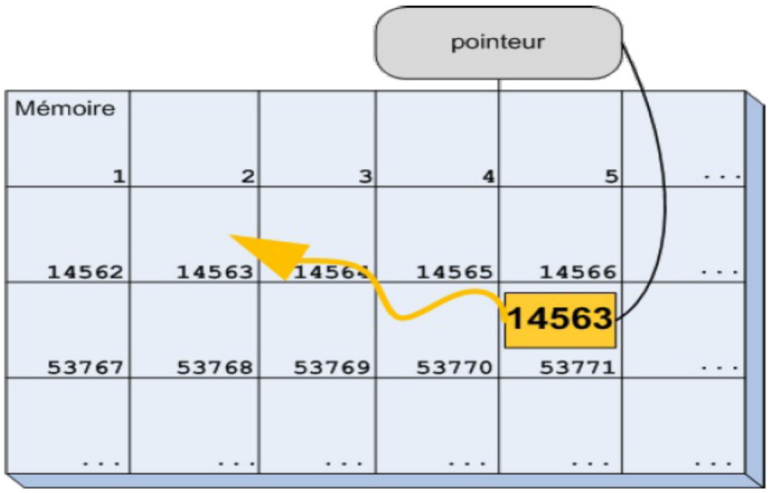
- **Libérer la mémoire**

Une fois que l'on a plus besoin de la case mémoire, il faut la rendre à l'ordinateur. Cela se fait via l'opérateur **delete**.

Code : C++

```
int *pointeur(0);
pointeur = new int;
delete pointeur; //On libère la case mémoire
```

La case est alors rendue à l'ordinateur qui pourra l'utiliser pour autre chose. Le pointeur, lui, existe toujours. Et il pointe toujours sur la case mais vous n'avez **plus le droit** de l'utiliser.



L'image est très parlante. Si l'on suit la flèche, on arrive sur une case qui ne nous appartient pas. Il faut donc impérativement empêcher ça. Imaginez que cette case soit soudainement utilisée par un autre programme ! Vous risqueriez de modifier les variables de cet autre programme.
Il est donc **essentiel** de supprimer cette "flèche" en mettant le pointeur à l'adresse 0 après avoir utilisé **delete**. Ne pas le faire est une source très courante de plantage des programmes.

Code : C++

```
int *pointeur(0);
pointeur = new int;
delete pointeur; //On libère la case mémoire
pointeur = 0; //On indique que le pointeur ne pointe vers plus rien
```

Exemple

Terminons cette section avec un exemple complet : un programme qui demande son âge à l'utilisateur et qui l'affiche en utilisant un pointeur.

Code : C++

```
#include <iostream>
using namespace std;
int main()
{
int* pointeur(0);
pointeur = new int;
```

```
cout << "Quel est votre age ? ";
cin >> *pointeur; //On écrit dans la case mémoire pointée par
le pointeur 'pointeur'
cout << "Vous avez " << *pointeur << " ans." << endl; //On
utilise à nouveau *pointeur
delete pointeur; //Ne pas oublier de libérer la mémoire
pointeur = 0; //Et de faire pointer le pointeur vers rien
return 0;
}
```

Ce programme est plus compliqué que sa version sans allocation dynamique ! C'est vrai. Mais on a le contrôle complet sur l'allocation et la libération de la mémoire.

Dans la plupart des cas, ce n'est pas utile de le faire. Mais vous verrez plus tard que, pour faire des fenêtres, la bibliothèque **Qt** utilise beaucoup **new** et **delete**. On peut ainsi maîtriser précisément quand une fenêtre est ouverte et quand on la referme par exemple.

Problème1 (les tableaux) :

Exemple :

Un programme qui calcule la moyenne de notes.

Code : C++ - Calcul de la moyenne des notes

```
#include <iostream>
using namespace std;
int main()
{
int const nombreNotes(6);
double notes[nombreNotes];
notes[0] = 12.5;
notes[1] = 19.5; //Bieeeen !
notes[2] = 6.; //Pas bien !
```

```
notes[3] = 12;
notes[4] = 14.5;
notes[5] = 15;
double moyenne(0);
for(int i(0); i<nombreNotes; ++i)
{
moyenne += notes[i]; //On additionne toutes les notes
}
//En arrivant ici, la variable moyenne contient la somme des
notes (79.5)
//Il ne reste donc qu'a diviser par le nombre de notes
moyenne /= nombreNotes;
cout << "Votre moyenne est : " << moyenne << endl;
return 0;
}
```

Code : Console : Exécution

Votre moyenne est : 13.25

Calcule de la moyenne des notes

Soit un ensemble des notes des élevés donné. Pour calculer la moyenne, il nous faut additionner toutes les notes et ensuite diviser par le nombre de notes. Nous connaissons déjà le nombre de notes, puisque nous avons **la constante nombreNotes**. Il ne reste donc qu'à déclarer une variable pour contenir la moyenne.

Ecrire le programme qui permet de calculer la note moyenne

notes[0] = 12.5;notes[1] = 19.5;notes[2] = 6.;notes[3] = 12;notes[4] = 14.5;notes[5] = 15;notes[6] = 19;

Programme en c++ :

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

```
int main()
{
int const nombreNotes(7);
double notes[nombreNotes];
notes[0] = 12.5;
notes[1] = 19.5; //Bieeeen !
notes[2] = 6.; //Pas bien !
notes[3] = 12;
notes[4] = 14.5;
notes[5] = 15;
notes[6] = 19;
double moyenne(0);
for(int i(0); i<nombreNotes; ++i)
{
   moyenne += notes[i]; //On additionne toutes les notes
}//En arrivant ici, la variable moyenne contient la somme des notes (79.5)
//Il ne reste donc qu'a diviser par le nombre de notes
moyenne /= nombreNotes;
cout << "Votre moyenne est : " << moyenne << endl;
return 0;
}
```

Problème2 (les pointeurs) :

```
#include <iostream>

using namespace std;

int main()

{
int ageUtilisateur(16);
int *ptr(0);
```

```
ptr = &ageUtilisateur;
// obtenir l'adresse d'une variable
cout << "L'adresse de 'ageUtilisateur' est : " <<
&ageUtilisateur << endl;
cout << "La valeur de pointeur est : " << ptr << endl;
return 0;
}
```

Exécution

Code : Console

```
L'adresse de 'ageUtilisateur' est : 0x2ff18
La valeur de pointeur est : 0x2ff18
```

Travail demandé :

1-Ecrire l'algorithme qui permet de donner le programme ci-dessus qui permet de calculer la moyenne des notes

2-écrire le programme qui permet de calculer la moyenne des notes des étudiants suivants

notes[0] = 15;notes[1] = 15;notes[2] = 6.;notes[3] = 12;notes[4] = 14;notes[5] = 15;notes[6] = 16; notes[7] = 11;notes[8] = 19.5;notes[9] = 6.;notes[10] = 12;notes[11] = 15;notes[12] = 15.5;notes[13] = 17; notes[14] = 12;notes[15] = 10

Solution du travail demandé

Code c++ :

```
#include <iostream>

using namespace std;
int main()
{
int const nombreNotes(16);
double notes[nombreNotes];
notes[0] = 15;
notes[1] = 15; //Bieeeen !
notes[2] = 6.; //Pas bien !
notes[3] = 12;
notes[4] = 14;
notes[5] = 15;
notes[6] = 16;
notes[7] = 11;
notes[8] = 19.5;
notes[9] = 6;
notes[10] = 12;
notes[11] = 15;
notes[12] = 15.5;
notes[13] = 17;
notes[14] = 12;
notes[15] = 10;

double moyenne(0);
for(int i(0); i<nombreNotes; ++i)
{
   moyenne += notes[i]; //On additionne toutes les notes
```

```
}//En arrivant ici, la variable moyenne contient la somme des notes (211)
//Il ne reste donc qu'a diviser par le nombre de notes
moyenne /= nombreNotes;
cout << "Votre moyenne est : " << moyenne << endl;
return 0;
}
```

TRAVAUX PRATIQUES TP N°5

LES FONCTIONS

1. INTRODUCTION

Une fonction est un bloc d'instructions éventuellement paramétré par un ou plusieurs arguments et pouvant fournir un résultat nommé souvent «valeur de retour ». On distingue la définition d'une fonction de son utilisation, cette dernière nécessitant une déclaration ci-dessous.

Une fonction est un morceau de code qui accomplit une tâche particulière. Elle reçoit des données à traiter, effectue des actions avec et finalement renvoie une valeur.

Les données entrantes s'appellent les arguments et on utilise l'expression valeur retournée pour les éléments qui sortent de la fonction.

Lorsqu'on a un ensemble de lignes de code qui doivent être exécutées à différents endroits dans un programme, au lieu de réécrire les mêmes lignes de code, il est intéressant de créer des fonctions.

Au lieu d'écrire une fonction **main()** de **500** lignes, il est préférable de créer **25** fonctions de **20** lignes

• on structure le programme.

• il est plus facile de tester chaque fonction.

Dans ce TP, le but est d'apprendre à utiliser les fonctions pour permettre une approche modulaire de la programmation avec C++.

2. Rappel théorique

1. Définir une fonction

La définition d'une fonction se présente comme dans cet exemple :

Syntaxe :

Toutes les fonctions ont la forme suivante :

Code : C++

```
type nomFonction(arguments)
{
//Instructions effectuées par la fonction
}
```

On retrouve les trois éléments de la fonction :

1. Le premier élément est le **type de retour**. Il permet d'indiquer le type de variable renvoyé par la fonction. Si votre fonction doit renvoyer du texte, alors ce sera string, si votre fonction effectue un calcul, alors ce sera int ou double.
2. Le deuxième élément est le **nom de la fonction**, exemple la fonction **main.**
3. Entre les parenthèses, on trouve la **liste des arguments** de la fonction. Ce sont les données avec lesquelles la fonction va travailler. Il peut y avoir un argument (comme pour sqrt()), ou aucun

argument (comme pour main()).

4. Finalement, il y a des **accolades** qui délimitent le contenu de la fonction. Toutes les opérations qui seront effectuées se trouvent entre les deux accolades.

Exemple d'une fonction

```
#include <iostream>

using namespace std;
void b()
{
cout<<"Bonjour"<<endl;
}
int main()
{
cout<<"COUCOU1"<<endl;
b();
cout<<"COUCOU2"<<endl;
b();
b();
cout<<"COUCOU3"<<endl;
b();
return 0;

}
```

Dans ce programme, on a créé une fonction b qui se contente d'afficher "Bonjour" à l'écran. La fonction b est précédée du type void : cela signifie que la fonction ne renvoie aucune valeur au programme appelant.

• Le programme principal (la fonction main()) affiche "COUCOU1" à l'écran, ensuite appelle la fonction b, affiche le message "COUCOU2" à l'écran, appelle ensuite 2 fois la fonction b, affiche le message "COUCOU3" et appelle une dernière fois la fonction b.

Code console (Exécution de l'exemple)

Lorsqu'on exécute le programme voici ce qu'on obtient à l'écran :

```
COUCOU1
Bonjour
COUCOU2
Bonjour
Bonjour
COUCOU3

Bonjour
```

- **Appel à une fonction**

Lors de l'appel de la fonction, le programme exécute la totalité des instructions du corps de la fonction, puis reprend le programme juste après l'appel de la fonction.

Exemple

Code : C++

```
#include <iostream>
using namespace std;
int ajouteDeux(int nombreRecu)
{
int valeur(nombreRecu + 2);
return valeur;
}
int main()
```

```
{
int a(2),b(2);
cout << "Valeur de a : " << a << endl;
cout << "Valeur de b : " << b << endl;
b = ajouteDeux(a); //Appel de la fonction
cout << "Valeur de a : " << a << endl;
cout << "Valeur de b : " << b << endl;
return 0;
  }
```

Code : Console

```
Valeur de a : 2
Valeur de b : 2
Valeur de a : 2
Valeur de b : 4
```

Après l'appel à la fonction, la variable **b** a été modifié. Tout fonctionne donc comme annoncé.

- **Fonction à plusieurs paramètres**

On n'est pas encore au bout de nos peines. Il y a des fonctions qui prennent plusieurs paramètres.

Pour passer plusieurs paramètres à une fonction, il faut les séparer par des virgules.

Exemple1:

Code : C++

```
int addition(int a, int b)
{
return a+b;
}
double multiplication(double a, double b, double c)
{
return a*b*c;
}
```

La première de ces fonctions calcule la somme des deux nombres qui lui sont fournis alors que la deuxième calcule le produit des trois nombres reçus.

- **Fonction sans arguments**

A l'inverse, il est aussi possible de créer des fonctions sans arguments. Il faut simplement ne rien écrire entre les parenthèses.

On peut imaginer plusieurs scénarios, mais pensez par exemple à une fonction qui demande à l'utilisateur d'entrer son nom. Elle n'a pas besoin de paramètres.

Code : C++

```
string demanderNom()
{
cout << "Entrez votre nom : ";
string nom;
cin >> nom;
return nom;

}
```

- **Des fonctions qui ne renvoient rien**

Il est aussi possible d'écrire des fonctions qui ne renvoient rien. Enfin presque. Rien ne ressort de la fonction, mais quand on la déclare, il faut quand même indiquer un type. On utilise le "type" **void**, ce qui signifie **néant** en anglais. Ça veut tout dire, il n'y a réellement rien qui ressort de la fonction.

Exemple :

Code : C++ - Une fonction ne renvoyant rien

```
void direBonjour()
{
cout << "Bonjour !" << endl;
//Comme rien ne ressort, il n'y a pas de return !
}
int main()
{
```

```
direBonjour(); //Comme la fonction ne renvoie rien, on
l'appelle
//sans mettre la valeur de retour dans une
variable
return 0;
}
```

3-Travail demandé :

3.1. Travail demandé 1

Commençons par une fonction basique. Une fonction qui reçoit un nombre entier, ajoute 2 à ce nombre et le renvoie.

3.2. Travail demandé 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).

Solution du travail demandé

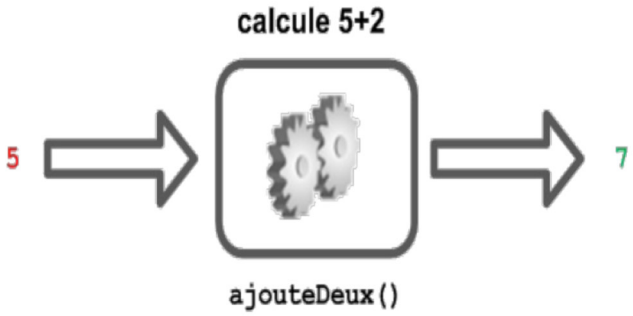
Déclaration de la fonction

```
int ajouteDeux(int nombreRecu)

{
int valeur(nombreRecu + 2); //On cree une case en memoire.
//On prend le nombre recu en argument, on y ajoute 2.
//Et on met tout ça dans la memoire.
return valeur; //On indique que la valeur qui sort de la fonction
//est la valeur de la variable 'valeur'

}
```

On déclare une fonction nommée **ajouteDeux** qui va recevoir un nombre entier en argument et qui, une fois qu'elle aura terminé, va recracher un autre nombre entier.



Le **return** de l'avant-dernière ligne indique quelle valeur va ressortir de la fonction. En l'occurrence, c'est la **valeur** de la variable valeur qui va être renvoyée.

Appeler une fonction

```
#include <iostream>
using namespace std;
int ajouteDeux(int nombreRecu)
{
int valeur(nombreRecu + 2);
```

```
return valeur;
}
int main()
{
int a(2),b(2);
cout << "Valeur de a : " << a << endl;
cout << "Valeur de b : " << b << endl;
b = ajouteDeux(a); //Appel de la fonction
cout << "Valeur de a : " << a << endl;
cout << "Valeur de b : " << b << endl;
return 0;
}
```

Exécution du programme

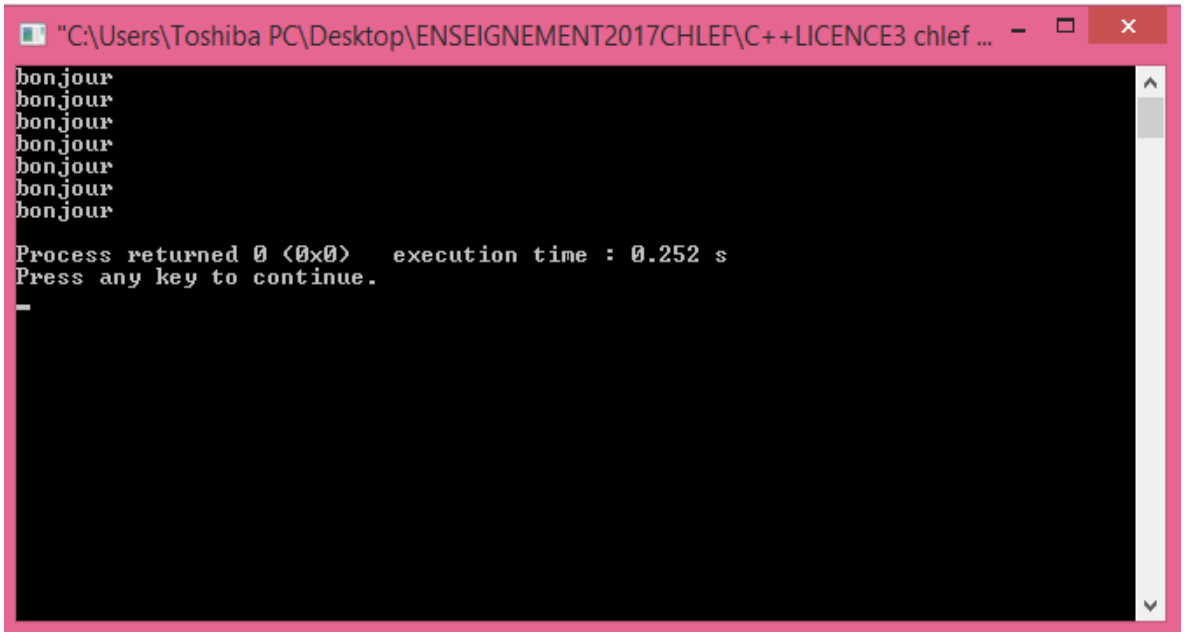
Code : Console

```
Valeur de a : 2
Valeur de b : 2
Valeur de a : 2
Valeur de b : 4
```

3.2. L'énoncé ne précisant rien, nous utiliserons une transmission d'arguments par valeur. Comme on n'impose pas d'ordre aux définitions des différentes fonctions dans le fichier source, on déclarera systématiquement toutes les fonctions utilisées.

Programme en C++ :

```
#include <iostream>
using namespace std ;
void f1 (void)
{ cout << "bonjour\n" ;
}
void f2 (int n)
{ int i ;
for (i=0 ; i<n ; i++)
cout << "bonjour\n" ;
}
int f3 (int n)
{ int i ;
for (i=0 ; i<n ; i++)
cout << "bonjour\n" ;
return 0 ;
}
main()
{ void f1 (void) ;
void f2 (int) ;
int f3 (int) ;
f1 () ;
f2 (3) ;
f3 (3) ;
}
```



Travaux Pratiques TP N°6

Fichiers

1. INTRODUCTION

Un programme a en général besoin :

- De lire des données (texte, nombres, images, sons, mesures, …)
- De sauvegarder des résultats (texte, nombres, images, sons, signaux générés, …)

Donc cela se fait en lisant et en écrivant dans des fichiers.

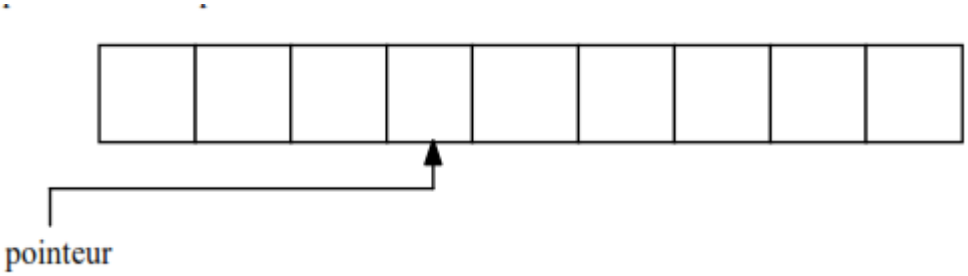
2. Pour manipuler un fichier on utilise un **pointeur** sur une donnée• Pour manipuler un fichier, on utilise un **pointeur** sur une donnée spécifique dont le type est **FILE** (structure prédéfinie que nous n'avons pas besoin de connaître précisément) :

FILE *fichier

3. La variable **fichier** contiendra l'adresse en mémoire du début du fichier.

Définition :

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.



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.

Fichiers textes ou binaires

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

• les fichiers textes qui sont des fichiers lisibles par un simple éditeur de texte.

• les fichiers binaires dont les données correspondent en général à une copie bit à bit du contenu de la RAM. Ils ne sont pas lisibles avec un éditeur de texte.

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

1. fichiers 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 …

3- fichiers 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 …

1. **cstdio ou fstream**

Il existe principalement 2 bibliothèques standard pour écrire des fichiers :

• **cstdio** qui provient en fait du langage C.

• **fstream** qui est typiquement C++.

Utilisation de cstdio

La fonction FILE * fopen(const char * filepath, char * mode)

Cette fonction permet d'ouvrir un fichier en lecture ou en écriture. Le paramètre *filepath* est un tableau de char contenant le chemin du fichier sur lequel on souhaite travailler. Le paramètre *mode* indique le mode d'ouverture de

filepath **:** lecture ou écriture, texte ou binaire.

4-Opérations possibles avec les fichiers

1. Créer
2. Ouvrir

3. Lire
4. Ecrire
5. Détruire
6. Renommer
7. Fermer.

La plupart des fonctions permettant la manipulation des fichiers sont rangées dans la bibliothèque standard **STDIO.H**.

Ces fonctions sont très nombreuses. Seules quelques-unes sont présentées ici.

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).

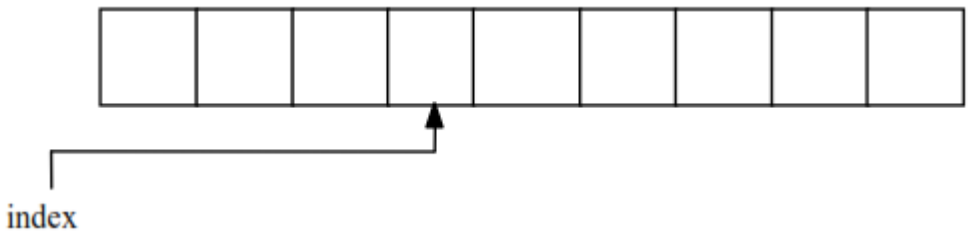
Lorsque la manipulation est terminée, il faut fermer ce fichier et vérifier sa fermeture effective.

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.

4-1 - 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 est la variable qui permettra de manipuler le fichier dans le programme.

4-2 - 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 »

1. 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.

2. 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).

Exemple

Code C++ :

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

4-3 - Fermeture

On utilise la fonction de prototype

```
int fclose(FILE *f);
```

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

Exemple :

Code C++

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

4-4 - Destruction :

On utilise la fonction de prototype

```
int remove(char *nom);
```

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

Exemple :

Code C++

```
int x ;
        x = remove("a :\\toto.dat") ;
    if (x == 0) cout << « Fermeture OK : » ;
```

```
else cout << « Problème à la fermeture : » ;
```

4-5 - 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 :

Code C++

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

5-MANIPULATIONS DES FICHIERS TEXTES

5-1- 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) ;**

5-2- 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.

3. 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 estatteinte. Via une conversion automatique de type, on obtient le caractère.
4. 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 :

Codes C++

```
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.

6-MANIPULATIONS DES FICHIERS BINAIRES

1. La fonction **int feof(FILE *index)** retourne 0 tant que la fin du fichier n'est pas atteinte.
2. La fonction **int ferror(FILE *index)** retourne 1 si une erreur est apparue lors d'une manipulation de fichier, 0 dans le cas contraire.
3. 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).

Application:

taille_bloc = 4 (taille d'un entier en C++), nb_bloc=3, écriture de 3 entiers.

```
int tab[10] ;
fwrite(tab,4,3,index) ;
```

1. 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).

Travaux Pratiques TP N°7

Programmation orientée objet en C++

Classes, Méthodes particulières (constructeurs, destructeurs…), Héritage

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)
- l'efficience (performance en termes de vitesse d'exécution et de consommation mémoire)

Les langages évolués de type C ou PASCAL, reposent sur le principe de la programmation structurée (algorithmes + structures de données)

Programmes = algorithmes + structures de données

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.

Méthodes + Données = Objet

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.

2. 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.
3. 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 appliqués. 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()

La méthode size() permet de connaître la longueur de la chaîne actuellement s tockée 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 :
Code : C++

```
int main()
{
   string maChaine("souad !");
   cout << "Longueur de la chaine : " << maChaine.size();
   return 0;
}
```

Code : Console

```
Longueur de la chaine : 7
```

Exemple : Objet Cercle

```
{Rayon, abs, ord de type réel // 3 Champs de type réels
Méthode périmètre de type réel   // fonction qui calcul le périmètre du cercle
{
Périmètre= 2*3,14*rayon
}
Méthode surface de type réel   // fonction qui calcul la surface du cercle
{
surface= 2*3,14*rayon*rayon
}
}
```

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 public 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 :

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

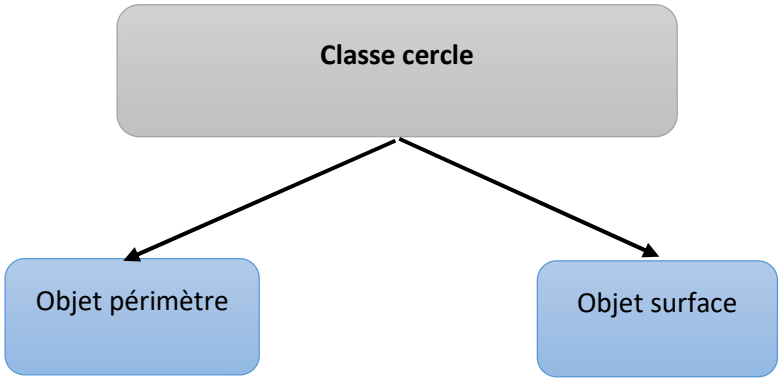


Figure : classe cercle

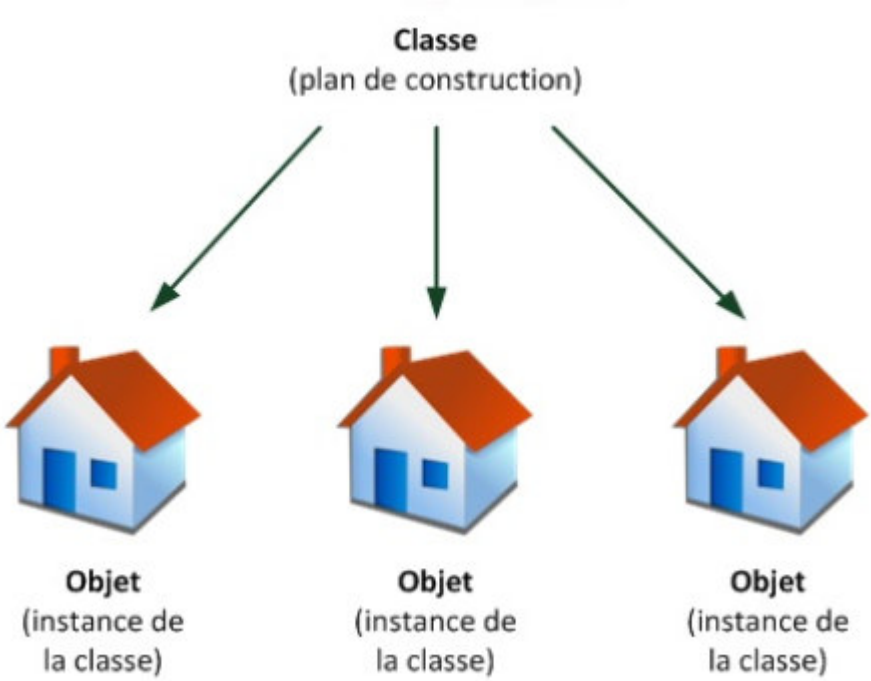


Figure : Classe plan de construction

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*)

Exemple : Classe Cercle

```
{Rayon, abs, ord de type réel // 3 Champs de type réels
Méthode périmètre de type réel   // fonction qui calcul le périmètre du cercle
{
Périmètre= 2*3,14*rayon
}
Méthode surface de type réel   // fonction qui calcul la surface du cercle
{
surface= 2*3,14*rayon*rayon
}

}
```

3. FONDAMENTAUX DE LA POO

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

1. Héritage
2. Encapsulation
3. Polymorphisme

3-1- NOTION D'HERITAGE :

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.

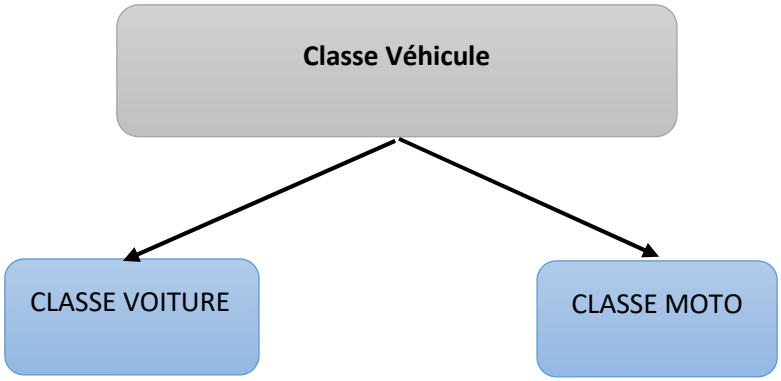
De cette manière il est possible de récupérer des librairies de classes, qui constituent une base, pouvant être spécialisées à loisir. Une particularité de l'héritage est qu'un objet d'une classe dérivée est aussi du type de la classe mère.

Dans la classe fille, on trouve :

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

Application :

Les classes voiture et moto décrivent de la classe Véhicule.



Les classes **Voiture** et **moto** héritent les propriétés (champs) et les méthodes de la classe **Véhicule**.

Classe **véhicule**

Champs : type, puissance, couleur ;

Méthodes : rouler ( ) ;

Garer ( ) ;

Classe **voiture**

Champs : volant, vitesse ;

Méthodes : reculer ( ) ;

Garer ( ) ;

Classe **moto**

Champs : guidon, chaine ;

Méthodes : se-faufiler ( ) ;

Remarques sur l'exemple :

La classe Voiture :

1. Conserve les champs Types, couleur et puissance de Véhicule
2. Possède les nouveaux champs **volant** et **vitesse**
3. Conserve la méthode **rouler ( )** de véhicule.
4. Surcharge la méthode Garer de Véhicule
5. Implémente la méthode **reculer ( )**

La classe Moto :

6. Conserve les propriétés Types, couleur et puissance de Véhicule
7. Possède le nouveau champ **Guidon**
8. Conserve la méthode **rouler ( )** de véhicule.
9. Conserve la méthode **Garer ( )** de véhicule.
10. Implémente la méthode **se-faufiler ( )**

Par ce moyen on crée une hiérarchie de classes de plus en plus spécialisées. 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 développer des librairies de classes, qui constituent, une base, pouvant être spécialisées et réutilisables.

1. A-HIERARCHIE DES CLASSES

Il est possible de représenter sous forme de hiérarchie de classes, parfois appelée arborescence de classes, la relation de parenté qui existe entre les différentes classes. L'arborescence commence par une classe mère appelée **superclasse.** Puis les classes dérivées (classe fille ou sous-classe) deviennent de plus en plus spécialisées. Ainsi, on peut généralement exprimer la relation qui lie une classe fille à sa mère par la phrase « est un » (de l'anglais « is a »).

Exemple :

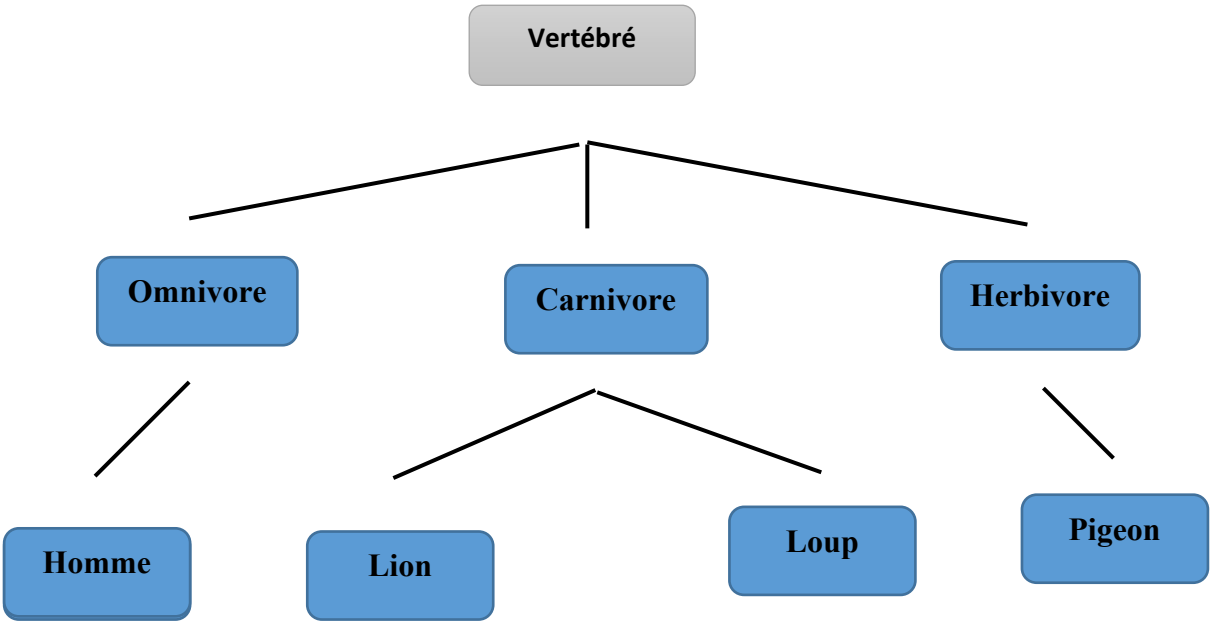


Figure : Hiérarchie de classes

1. On dit que l'homme est un omnivore vertébré.
2. Le lion est un carnivore vertébré.
3. Le pigeon est un herbivore vertébré.

4. B-HIERARCHIE MULTIPLE

Certains langages orientés objets, tels que le C++ , permettent de faire de l'héritage multiple, ce qui signifie qu'ils offrent la possibilité de faire hériter une classe de deux superclasses.

Ainsi, cette technique permet de regrouper au sein d'une seule et même classe les attributs et méthodes de plusieurs classes.

Exemple :

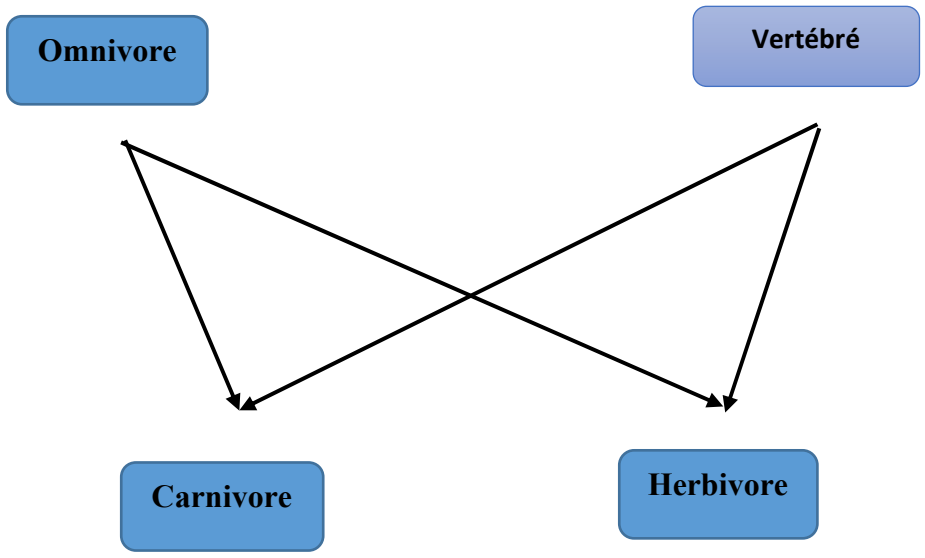


Figure : Hiérarchie multiple de classes

Les classes filles Carnivore et Herbivore Héritent à la fois des deux superclasses Omnivore est Vertébré.

3.2 NOTION D'ENCAPSULATION

En ce que l'on pourrait qualifier de P.O.O. « pure », on réalise ce que l'on nomme une **encapsulation des données**. Cela signifie qu'il n'est pas possible d'agir directement sur les données d'un objet ; il est nécessaire de passer par l'intermédiaire de ses méthodes, qui jouent ainsi le rôle d'interface obligatoire. On traduit parfois cela en disant que l'appel d'une méthode est en fait l'envoi d'un «Message» à l'objet.

Le grand mérite de l'encapsulation est que, vu de l'extérieur, un objet se caractérise uniquement par les spécifications de ses méthodes, la manière dont sont réellement implantées les données étant sans importance. On décrit souvent une telle situation en disant qu'elle réalise une « abstraction des données » (ce qui exprime bien que les détails concrets d'implémentation sont cachés). À ce propos, on peut remarquer qu'en programmation structurée, une procédure

pouvait également être caractérisée (de l'extérieur) par ses spécifications, mais que, faute d'encapsulation, l'abstraction des données n'était pas réalisée.

L'encapsulation des données présente un intérêt manifeste en matière de qualité de logiciel. Elle facilite considérablement la maintenance : une modification éventuelle de la structure des données d'un objet n'a d'incidence que sur l'objet lui-même ; les utilisateurs de l'objet ne seront pas concernés par la teneur de cette modification (ce qui n'était bien sûr pas le cas avec la programmation structurée).

De la même manière, l'encapsulation des données facilite grandement la réutilisation d'un objet.

Derrière ce terme se cache le concept même de l'objet : réunir sous la même entité les données et les moyens de les gérer, à savoir les champs et les méthodes.

L'**encapsulation** introduit donc une nouvelle manière de gérer des données. Il ne s'agit plus de déclarer des données générales puis un ensemble de procédures et fonctions destinées à les gérer de manière séparée, mais bien de réunir le tout sous le couvert d'une seule et même entité.

Si l'**encapsulation** est déjà une réalité dans les langages procéduraux (comme le Pascal non objet par exemple) au travers des unités et autres librairies, il prend une toute nouvelle dimension avec l'**objet**.

En effet, sous ce nouveau concept se cache également un autre élément à prendre en compte : pouvoir masquer aux yeux d'un programmeur extérieur tous les rouages d'un objet et donc l'ensemble des procédures et fonctions destinées à la gestion *interne* de l'objet, auxquelles le programmeur final n'aura pas à avoir accès. L'encapsulation permet donc de masquer un certain nombre de champs et méthodes tout en laissant visibles d'autres champs et méthodes.

Pour conclure, l'encapsulation permet de garder une cohérence dans la gestion de l'objet, tout en assurant l'intégrité des données qui ne pourront être accédées qu'au travers des méthodes visibles.

L'**encapsulation** permet de définir des niveaux de visibilité des éléments de la classe. Ces niveaux de visibilités définissent les droits d'accès aux données selon que l'on y accède par une méthode de la classe elle-même, d'une classe héritière, ou bien d'une classe quelconque.

Il existe trois niveaux de visibilité :

1. Publique :

Les fonctions de toutes les classes peuvent accéder aux données ou aux méthodes d'une classe définie avec le niveau de visibilité public. Il s'agit du niveau de protection de données.

2. Privée :

L'accès aux données est limité aux méthodes de la classe elle-même. Il s'agit du niveau de protection des données le plus élevé.

3. Protégée :

L'accès aux données est réservé aux fonctions des classes héritières, càd par les fonctions membres de la classe ainsi des classes dérivées.

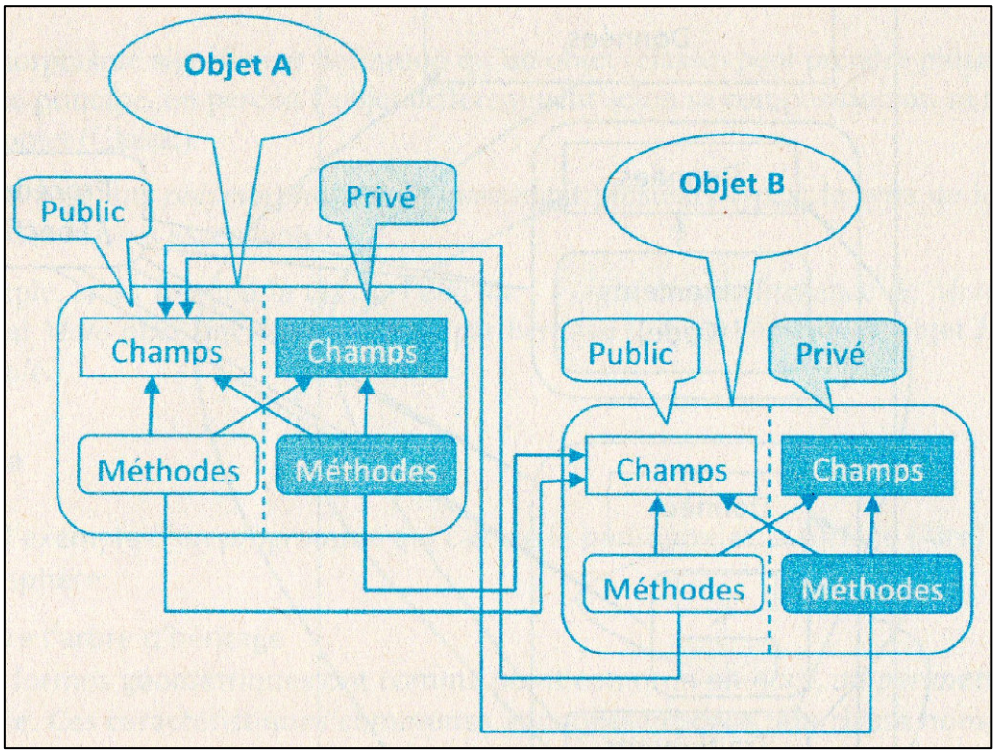


Figure : Accès aux membres publics entre objets

Remarque :

Sur le schéma de la figure, les méthodes de l'objet A ne peuvent accéder qu'a ses propres champs ou/et aux champs publics de l'objet B.

Les méthodes de l'objet B ne peuvent accéder qu'a ses propres champs ou/ et aux champs publics de l'objet A.

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.

Application :

Prenons l'exemple d'un programme qui calcule le périmètre et la surface (aire) d'in cercle et d'une sphère.

1-Déduire l'arbre d'héritage les deux formes géométriques ont comme point commun un nom, un périmètre, un aire et un volume. Ces caractéristiques communes apparaissent dans une classe nommée figure se trouvant à la racine de l'arbre d'héritage. Par la suite, pour déterminer le périmètre et l'aire d'un cercle, on doit connaitre son rayon, il en va de même pour calculer l'aire et le volume d'une sphère. Cette information supplémentaires particulière à chaque figure conduit à la définition de deux classes distinctes : cercle descendante de la classe figure, et sphère, la descendante de la classe cercle. La classe cercle contient comme donnée la longueur du rayon.

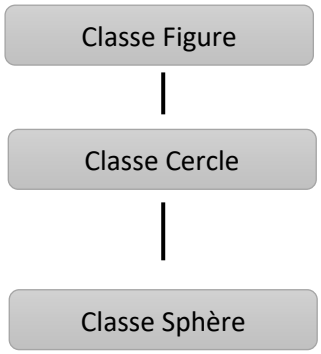


Figure : Arbre d'héritage de la Classe Figure.

2-Determiner les méthodes de chaque classe

La classe Figure doit permettre l'affichage de l'information qu'elle contient

Classe Figure
Champs :
nomFigure, Aire, Perimetre, Volume ;
Méthodes :
afficherInfo () ;
Classe Cercle
Champs :
Rayon ;
Méthodes :
obtenirRayen ( ) ;
calculerperimetre ( ) ;
calculerAire( ) ;

Classe Sphère
Champs :
Méthodes :
calculerAire( ) ;
calculerVolume ( ) ;

Remarques

L'appel de la fonction calculer Aire ( ) est identique pour un objet de la classe Cercle et pour un objet de la classe Sphère, même si les calculs de l'aire d'un cercle et de l'aire d'une sphère sont différents c'est le pricipe du polymorphisme.

4-METHODES 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.

1. Les constructeurs.
2. 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 créer 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 une fois fini l'utilisation de l'exemplaire et récupère la zone mémoire occupée.

Bibliographie

1- A. Azough, Cours du Langage C++, Université Sidi Mohamed Ben Abdellah, Faculté des sciences, Maroc, 2017
2- B. Stroustrup, Le Langage C++, Édition Addison-Wesley 2000.
3- Bjarne Stroustrup, Marie-Cécile Baland, Emmanuelle Burr, Christine Eberhardt, Programmation : Principes et pratique avec C++, Edition Pearson 2012.
4- Bartjan van Tent, COURS entrées, sorties, fichiers, en C++ Université Paris-Sud – Orsay, L3 et Magistère 1ère année de Physique Fondamentale, 2015
5- Belaid, Programmation en C++,COURS +TD+TP , Edition pages bleues 2016.
6- B. Stroustrup, M. Baland, E. Burr, C. Eberhardt, Programmation : Principes et pratique avec C++, Edition Pearson 2012.
7- C. Delannoy, Programmer en langage C++, Edition Eyrolles 2000.
8- Claude Delannoy, Programmer en langage C++, Edition Eyrolles 2000.
9- Cours de Langage C , Lecture & écriture dans des fichiers, Institut d'Optique Graduate School ,France
10- D. Claude Exercices en langage C++ Ed3. Eyrolles 2007
11- Delannoy Claude, Exercices en langage, C++ » Ed3. Eyrolles 2007
12- F. MALIKI , Cours Algorithmique 2 : Techniques de programmation orientée objet, EPST Tlemcen 2014
13- F. Drouillon, Du C au C++ - De la programmation procédurale à l'objet, Eni; Édition :2e édition 2014.
14- Jean-Cédric Chappelier, Florian Seydoux, C++ par la pratique. Recueil d'exercices corrigés et aide-mémoire, PPUR Édition : 3e édition 2012.
15- Jean-Michel Léry, Frédéric Jacquenot, Algorithmique, applications aux langages C, C++ en Java Edition Pearson, 2013.
16- Joëlle MAILLEFERT, COURS et TP DE LANGAGE C++,IUT de CACHAN, Département GEII 2, Université Paris-Sud.
17- J. Chappelier, F. Seydoux, C++ par la pratique. Recueil d'exercices corrigés et aide-mémoire, PPUR Édition : 3e édition 2012.
18- J. Léry, F. Jacquenot, Algorithmique, applications aux langages C, C++ en Java Edition Pearson, 2013.
19- Joëlle Maillefert, Cours et TP de langage C++, chapitre 1 : Elément de langage C++, IUT DE CACHAN, Département de GEII2.
20- Karl Tombre, Une courte introduction à C++ » École des Mines de Nancy,Version 1.0 ,Octobre 1999
21- Kris Jamsa, Lars Klander, C++ La bible du Programmeur, Edition Eyrolles 2000.
22- Mathieu Nebra et Matthieu Schaller, Programmez avec le langage C++, www.siteduzero.com
23- M. Nebra , M. Schaller « Programmez avec le langage C++ » www.siteduzero.com
24- S. Laporte , Cours C++ : le fichiers, BTS IG 1 Lycée Louise Michel, 2015-2016
25- T. Redarce, N. Ducros, O. Bernard, Langage C/C++, De la syntaxe à la programmation orientée objet, T. Grenier, November 21, 2017
26- https://openclassrooms.com/courses/programmez-avec-le-langage-c/les-tableaux-5
27- http://www.fresnel.fr/perso/stout/langage_C/Chap_8_Tableaux_de_char.pdf

28- http://sylvestre.ledru.info/howto/docC/x3616.html
29- http://webcache.googleusercontent.com/search?q=cache:http://roger.astier.free.fr/app/app_cpp/cppEs.html
30- https://www.fortisfio.com/les-flux-dentrees-sorties/
31- https://www.youtube.com/watch?v=JZp_HXRpoXw
32- https://www.youtube.com/watch?v=ydjLAJ3gE5c