République Algérienne Démocratique et populaire
**M**inistère de l'**E**nseignement **S**upérieur et de la **R**echerche **S**cientifique

**UNIVERSITE DE GUELMA**

Faculté des Sciences et de la Technologie
**Département d'Électronique et Télécommunications**

الجمهـورية الجـزائرية الديمقـراطية الشعـبية
وزارة التـعليم العالي والبحث العلمي

**جـامعة قالمـة**

كلية العلوم و التكنولوجيا
قسم الإلكترونيك والاتصالات
السلكية و اللاسلكية

# Programmation orientée objet en C++

**Cours et Travaux pratiques**

Préparé par :
**Hakim Doghmane**

Année universitaire : 2019-2020

# Avant-propos

Ce polycopié de cours "**Programmation orientée objet en C++**" s'adresse aux étudiants de la première année Master des parcours "**Instrumentation**", "**Réseaux et Télécommunications**" et "**Systèmes des Télécommunications**". Il est constitué de deux grandes parties. La première partie est consacrée à la description des fondements théoriques de la programmation orientée objet en C++. Elle est composée de quatre chapitres:

- Introduction à la programmation orientée objet
- Notions de base
- Classes et objets
- Héritage et polymorphisme

La deuxième partie de ce polycopié est destinée au travaux pratiques. Elle est basée principalement sur quatre axes:

- TP1: Maîtrise d'un compilateur C++.
- TP2: Programmation C++.
- TP3: Classes et objets.
- TP4: Héritage et polymorphisme.

L'objectif principal de ce polycopié est de mettre à la disposition des étudiants un support de cours avec des travaux pratiques afin qu'ils puissent apprendre les fondements de base de la programmation orientée objet ainsi que la maitrise des techniques de conception des applications en langage C++.

**Pré-requis:** Les connaissances préalables recommandées sont l'algorithmique et la programmation procédurale entre autre la programmation en C.

**Compte rendu du TP:**

Chaque binôme doit remettre un compte-rendu après avoir réalisé toutes les manipulations de TP en question. Le rapport doit contenir:

- Un rappel théorique.
- Réponses aux questions de chaque manipulation.
- Commentaires et analyse des résultats obtenus.

Certains exemples et exercices de ce polycopié ont été réalisés et testés avec **Turbo Borland C++ 3.0** et le reste avec **Code::Blocks 20.03**.

**Mode d'évaluation:**

- Examen final: 60%
- Contrôle continu: 40%
    - Micro-interrogation: 15%
    - Devoir à domicile: 10%
    - Test final de TP: 15%

# Table des matières

## Partie I

# Partie II

# Introduction à la programmation orientée objet (POO)

## 1. Introduction

Depuis l'invention de l'ordinateur, de nombreuses approches de programmation ont été proposées. La principale motivation de chacune d'entre elles est le souci de gérer la complexité croissante des programmes qui restent fiables et maintenables. Avec l'avènement des ces langages tels que le C, la programmation structurée est devenue très populaire et a été la principale technique des années 1980. C'est un outil puissant qui permettait aux programmeurs d'écrire assez facilement des programmes modérément complexes.

Ainsi, le langage C a été crée en 1972 dans le laboratoire de "Bell Telephone (AT&T)" avec les travaux de Brian Kernighan et Dennis Ritchie. Il a été conçu à l'origine pour l'écriture du système d'exploitation UNIX (90-95% du noyau est écrit en C). Il a été normalisé en 1989 par le comité X3J11 de l'American National Standards Institute (ANSI).

Les langages C, Pascal, Fortran et autres langages similaires sont des langages de procédure. C'est-à-dire que chaque énoncé dans le langage indique à l'ordinateur de faire quelque chose : la lecture, le calcul et l'affichage du résultat. Un programme dans un langage procédural est une liste d'instructions.

Lorsque les programmes deviennent plus volumineux, une seule liste d'instructions devient difficile à gérer. Peu de programmeurs peuvent comprendre un programme contenant plus de quelques centaines d'instructions, à moins qu'il ne soit décomposé en unités plus petites. C'est pourquoi cette fonction a été adoptée comme un moyen de rendre les programmes plus compréhensibles. Un programme de procédure est divisé en fonctions ou chaque fonction (sous programme) a une interface (en-tête) et un objectif clairement défini (corps de la fonction).

La division d'un programme en fonctions est l'une des pierres angulaires de la programmation structurée. La programmation classique ou procédurale traite les programmes comme un ensemble de données sur lesquelles agissent des procédures. Les procédures sont les éléments actifs où les données deviennent des éléments passifs

qui traversent l'arborescence de la programmation procédurale en tant que flot d'informations.

Néanmoins, le problème avec le paradigme procédural est que cela conduit à un nombre encore plus grand de connexions entre les fonctions et les données. Ce qui rend la structure du programme difficile a conceptualiser et pénible à faire des modifications. Dans certains cas, une modification apportée à une donnée globale peut nécessiter la réécriture de toutes les fonctions qui accèdent à cette donnée. De plus, un autre problème lié au paradigme procédural est que l'agencement des données et des fonctions séparées ne permet pas de modéliser les choses dans le monde réel. Dans le monde physique, nous avons à faire à des objets tels que des personnes et des voitures. Ces objets ne sont pas comme des données et ils ne sont pas aussi comme des fonctions. Les objets complexes du monde réel ont à la fois des attributs et un comportement. C'est pourquoi, L'idée fondamentale de la **P**rogrammation **O**rientée **O**bjet (**POO**) est de combiner en une seule unité à la fois les données et les fonctions qui opèrent sur ces données. Une telle unité est appelée "**objet**". La POO implique des concepts qui sont nouveaux pour les programmeurs des langages traditionnels tels que les classes, l'héritage et le polymorphisme.

**2. Principe de la POO**

Le principal facteur de motivation dans l'invention de l'approche orientée objet est de supprimer certaines failles rencontrées dans l'approche procédurale. La POO traite les données comme un élément critique dans le développement du programme et ne leur permet pas de circuler librement dans le système. Elle lie plus étroitement les données à la fonction qui les exploite et les protège contre toute modification accidentelle provenant d'une fonction extérieure. La POO permet la décomposition d'un problème en un certain nombre d'entités appelées objets, puis construit les données et les fonctions autour de ces objets. L'organisation des données et des fonctions dans les programmes orientés objet est illustrée à la figure 1.1. Les données d'un objet ne sont accessibles que par les fonctions associées à cet objet. Cependant, les fonctions d'un objet peuvent accéder aux fonctions d'autres objets.

Aborder un problème de programmation en langage orienté objet, ne demande plus comment le problème sera divisé en fonctions, mais comment il sera divisé en objets.

Penser en termes d'objets, plutôt que de fonctions. Cela résulte de l'étroite correspondance entre les objets au sens de la programmation et les objets du monde réel.

La programmation orientée objet ne concerne pas en premier lieu les détails du fonctionnement du programme. Il s'agit plutôt de l'organisation générale du programme. Il y a trois concepts qui donnent toute sa puissance à la P.O.O :

- Concept de modélisation à travers la notion de classe et d'instanciation de ces classes.
- Concept d'action à travers la notion d'envoi des messages et des méthodes à l'intérieur des objets.
- Concept de construction par réutilisation et amélioration par l'utilisation de la notion d'héritage.

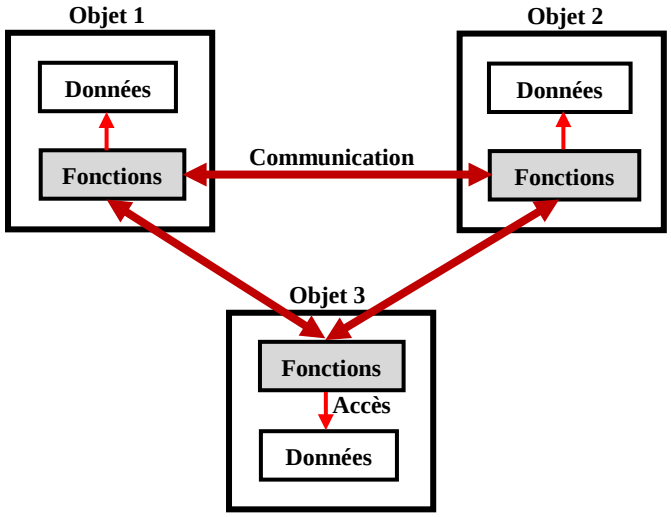



**Figure.1.1** : Organisation des données et des fonctions dans la POO.

La programmation orientée objet peut être définie comme une approche qui fournit un moyen d'organiser les programmes en créant une zone de mémoire partitionnée en données et des fonctions qui peuvent être utilisées comme modèles pour créer des copies de ces modules à la demande.

### 2.1. Concepts de base de la programmation orientée objet

Il est nécessaire de comprendre certains des concepts largement utilisés dans la programmation orientée objet. Il s'agit notamment de :

- Objets
- Classes

- Abstraction et encapsulation des données
- Héritage
- Polymorphisme

### 2.1.1. Objets

Les objets sont les entités d'exécutions de base dans un système orienté objet. Ils peuvent représenter une personne, un lieu ou tout autre élément que le programme doit traiter. Ils peuvent également représenter des données définies par l'utilisateur, telles que des vecteurs, des temps et des listes. Le problème de programmation est analysé en termes d'objets et de la nature de communication entre eux. Les objets du programme doivent être choisis de manière à correspondre étroitement aux objets du monde réel. Lorsqu'un programme est exécuté, les objets interagissent en s'envoyant des messages les uns aux autres. Par exemple, si "client" et "compte" sont deux objets dans un programme, alors un objet client peut envoyer un message à l'objet compte pour demander le solde bancaire. Chaque objet contient des données et un code permettant de manipuler ces données. Les objets peuvent interagir sans avoir à connaître les détails des données ou du code de l'autre. Il suffit de connaître le type de message accepté et le type de réponse renvoyée par les objets.

### 2.1.2. Classes

Une classe est une collection d'objets de type similaire. Par exemple, **le kiwi, la pomme et l'orange** sont des membres de la **classe fruit**. Les classes sont des types de données définis par l'utilisateur et se comportent comme les types intégrés d'un langage de programmation. Les objets ne sont que des variables de type classe. Si, par exemple, fruit a été défini comme une classe, alors l'instruction :

fruit kiwi;

va créer un objet kiwi appartenant à la classe fruit.

### 2.1.3. Abstraction et encapsulation des données

Le regroupement des données (attributs) et des fonctions en une seule unité (appelée classe) est connu sous le nom d'encapsulation. L'encapsulation des données est la caractéristique la plus frappante d'une classe. Les données ne sont pas accessibles au monde extérieur, et seules les fonctions qui sont encapsulées dans la classe peuvent y accéder. Ces fonctions assurent l'interface entre les données de l'objet et le programme.

Cette isolation des données de l'accès direct par le programme est appelée masquage des données. L'abstraction désigne l'acte de représenter des caractéristiques essentielles sans inclure les détails ou les explications de fond. Les attributs sont parfois appelés **données membres**, et les fonctions qui opèrent sur ces données sont appelées **méthodes** ou **fonctions membres**.

### 2.1.4. Héritage

L'héritage est le processus par lequel les objets d'une classe acquièrent les propriétés des objets d'une autre classe. Le principe, qui sous-tend ce type de division, est que chaque classe dérivée (sous classe ou classe spécialisée) partage des caractéristiques communes avec la classe mère (classe de base) dont elle est issue.

La classe spécialisée est intégralement cohérente avec la classe de base, mais comporte des informations supplémentaires (attributs, fonctions). Un objet de la classe spécialisée peut être utilisé partout où un objet de la classe de base est autorisé. Le schéma de la figure 1.2 se lit de bas en haut, c'est-à-dire que la sous classe "chat" hérite de la classe mère "mammifère". Elle possède toutes les caractéristiques de la classe mammifère.

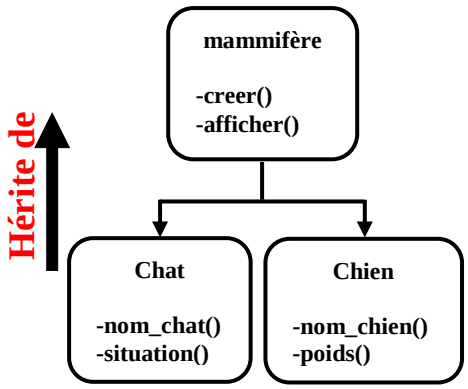



**Figure 1.2** : principe de l'héritage.

Donc, la sous classe 'chat" possèdera 4 fonctions membres :

- Deux fonctions membres de la classe dont elle hérite : creer() et afficher().
- Deux fonctions qui lui sont propres : nom_chat() et situation().

### 2.1.5. Polymorphisme

Le polymorphisme est un autre concept important de la POO. C'est un terme grec qui signifie la capacité de prendre plus d'une forme. Une opération peut présenter des comportements différents dans des cas différents. Ils dépendent des types de données

utilisées dans les opérations. Par exemple, considérons l'opération d'addition. Pour deux nombres, l'opération va générer une somme. Si les opérandes sont des chaînes de caractères, l'opération produira une troisième chaîne de caractères par concaténation.

La figure 1.3 illustre le fait qu'un seul nom de fonction (afficher()) peut être utilisé pour traiter différents types d'arguments. Il s'agit de quelque chose de similaire à un mot particulier ayant plusieurs significations différentes selon le contexte.

Le polymorphisme joue un rôle important en permettant à des objets ayant des structures internes différentes de partager la même interface externe. Il est largement utilisé dans la mise en œuvre de l'héritage.

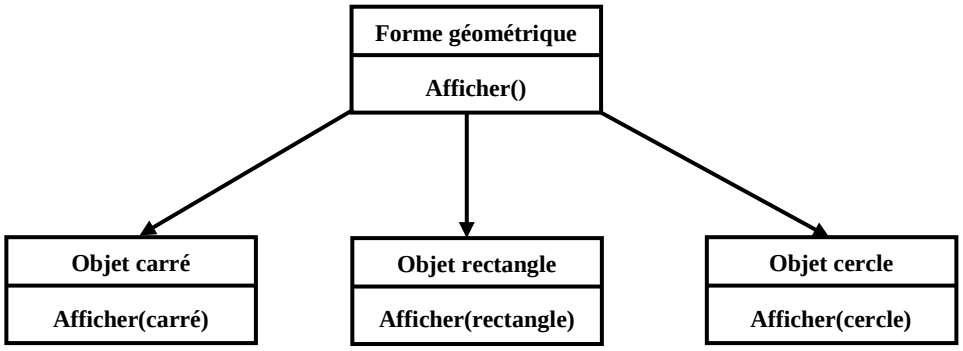



**Figure.1.3** : Polymorphisme.

## 2.2. Avantages de la POO

La POO offre plusieurs avantages tant au concepteur du programme qu'à l'utilisateur. L'orientation objet contribue à la résolution de nombreux problèmes liés au développement et à la qualité des produits logiciels. Les principaux avantages sont les suivants :

- Grâce à l'héritage, nous pouvons éliminer le code redondant et étendre l'utilisation des classes existantes.
- Nous pouvons construire des programmes à partir des modules de travail standard qui communiquent entre eux, plutôt que de devoir commencer à écrire le code à partir de zéro. Cela permet de gagner du temps de développement et d'augmenter la productivité.
- Le principe du masquage des données aide le programmeur à construire des programmes sûrs qui ne peuvent pas être envahis par le code d'autres parties du programme.

- Il est facile de partitionner le travail dans un projet basé sur des objets.
- Les systèmes orientés objet peuvent être facilement mis à niveau pour passer de petits à de grands systèmes.
- Les techniques de transmission des messages pour la communication entre les objets simplifient considérablement la description des interfaces avec les systèmes externes.
- La complexité des logiciels peut être facilement gérée.

**3. Définition du langage C++**

Le langage de programmation C++ est un langage faisant intervenir la P.O.O. En C++, le bloc composant les instructions du programme doit être placé entre accolades à la suite de la fonction principale main(). Dans tout programme C++, on utilise des instructions et des fonctions de la bibliothèque du C++. Ces fonctions sont contenues dans des fichiers en-tête dont l'extension est "**.h**". Parmi ces fichiers, on peut citer:

***iostream.h*** : qui contient entre autre les commandes d'entrées/sorties "cin()" et "cout()" pour respectivement, lire les données au clavier et afficher les résultats à l'écran.

***conio.h*** : contient entre autre les fonctions "clrscr()" et "gotoxy()" qui permettent respectivement d'effacer l'écran et de positionner le curseur à l'écran avant un affichage.

A chaque fois qu'on utilise une fonction, il faut inclure au début du programme le nom du fichier en-tête qui la contient. La structure d'un programme C++ se présente sous la forme suivante :

```
// Préprocesseurs
#include<iostream.h>
#include<conio.h>
#.....
// fonction principale
void main()
{
// déclaration des variables;
// instructions
}
```

Entre les deux accolades, on écrit les différentes déclarations et instructions du programme. Toute déclaration ou instruction se termine par un point virgule. Chaque programme C++ peut être divisé en 02 grandes parties :

1) Entête du programme.
2) Partie déclarative et instructions.

**Remarque**

Les fichiers en-tête standard (de la bibliothèque C++) seront introduits au début du programme après #include et entre **<.....>**. Alors que les fichiers en-tête crées par l'utilisateur seront aussi introduits au début du programme après #include et entre **"....."**.

### 3.1. L'alphabet du langage C++ et les règles de formation des mots

Un langage, et en particulier un langage de programmation est caractérisé par un alphabet à partir duquel, il est possible de former des mots. Dans un langage de programmation C++, on trouve :

Les lettres majuscules et minuscules (A …Z, et a …z).

Le caractère blanc (espace).

Les chiffres décimaux (0…9).

Les symboles spéciaux ({}, ;% ?+-=/…etc.).

A partir de l'alphabet, il est possible de construire les mots du langage. En programmation, le terme « mot » veut dire une suite de caractères de l'alphabet. Ces mots peuvent être dans l'une des catégories suivantes :

Les identificateurs.

Les variables.

Les constantes

Les chaînes de caractères.

Les mots clés (mots réservés).

#### 3.1.1. Identificateurs

Les identificateurs sont les noms donnés aux variables et aux fonctions d'un programme. C'est est une suite de caractères alphanumériques commençant obligatoirement par une lettre représentant une variable.

### 3.1.2. Variables

On appelle variable tout objet dont les valeurs sont soumises à une variation au cours du traitement. Les types de variables en C++ peuvent être classés dans différentes catégories, comme le montre la figure 1.4.

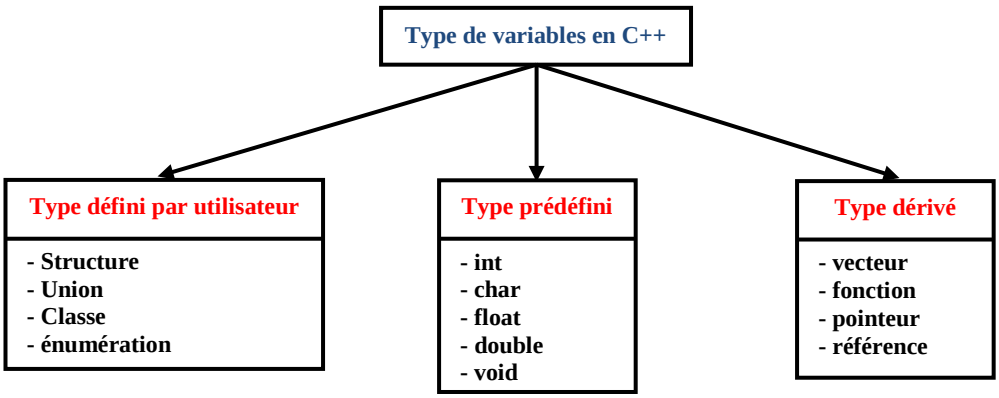



**Figure 1.4** : arborescence des types de données en C++.

### 3.1.3. Les constantes

Contrairement aux variables, les constantes ne changent pas de valeur tout au long du programme. Une constante doit être initialisée au moment de sa déclaration par le mot clé "const".

**Exemple 1.1:**

```
const int a=3;
const char b='s';
const char t[10]="master";
```

### 3.1.4. Les chaînes de caractères

En C++, les caractères doivent être encadrés par des apostrophes (‘ ’), tandis que les chaînes de caractères doivent être encadrées par deux guillemets (" ").

**Exemple 1.2:**

"Bonjour"  est une chaîne de caractères.

'A' est un caractère.

### 3.1.5. Les mots clés

Les mots clés (mots réservés) sont des mots propres pour le langage C++. On trouve les mots clés suivants : int, float, double, unsigned int, char, string, if, switch, … etc.

## 3.2. Concepts de base de la programmation en C++

### 3.2.1. Les opérations en C++

**a. Les opérations arithmétiques**

+ : Addition - : Soustraction * : Multiplication

/ : Division % : Le reste de la division entière

**b. Les opérations logiques**

! : Négation & : ET logique | : OU logique

**c. Les opérations de relation**

= = : Egale != : Différent de < : Inférieur à

> : Supérieur de <= : Inférieur ou égale >= : Supérieur ou égale

### 3.2.2. L'instruction d'affectation

L'instruction (=) permet d'affecter une valeur à une variable dans un instant donné.

**Exemple 1.3:**

Compteur=20 ;

A =B+C ;

### 3.2.3. L'instruction d'écriture (de sortie)

L'instruction de sortie est définie par le verbe « ECRIRE » (**cout** en C++) et par une suite des paramètres séparés par "<<". Les paramètres peuvent êtres :

**Des messages** à écrire sur l'écran (Chaîne de caractères entre deux guillemets).

cout<<"Bonjour Monsieur" ; → Bonjour Monsieur

**Des variables :** Afficher la valeur du variable

Si A=20 et de type entier dans un programme alors :

cout<<A ; → 20

**Exemple 1.4 :**

cout<<"la valeur de A est ="<<A ;

→ la valeur de A est =20

### 3.2.4. L'instruction de lecture (d'entrée)

L'instruction lire (**cin** en C++) permet de faire entrer des valeurs à travers le clavier par l'utilisateur.

**Syntaxe :**

LIRE(x) → cin>>x ;

### 3.2.5. Les commentaires

Dans le cas des programmes complexes, il est toujours possible d'ajouter des commentaires pour expliquer le rôle de chaque partie du programme. Ces commentaires ne seront pas compilés ni exécutés. Ils sont introduits entre antislashs et encadrés par des astérisques /*…….*/. Si le commentaire ne dépasse pas une ligne, on peut utiliser //.

**Exemple 1.5:**

```
#include <iostream.h>
#include <conio.h>
void main()
{
int x1,x2,p;
//c'est un programme qui calcul le produit de deux entiers
cout<<"x1=" ;
cin>>x1 ;
cout<<"x2=" ;
cin>>x2 ;
p=x1*x2 ;
cout<<"le produit de" <<x1<<" et"<<x2<<"est égale à"<<p ;
}
```

### 3.2.6. Caractères de commande précédés de " \" (Séquences d'échappement)

A partir des séquences d'échappement, l'impression et l'affichage du texte peuvent être contrôlées.

| | | | |
|---|---|---|---|
| **\a** | Sonnerie | \\ | Trait oblique |
| **\b** | Curseur arrière | **\?** | Point d'interrogation |
| **\t** | Tabulation | **\'** | Apostrophe |
| **\n** | Nouvelle ligne | \" | Guillemets |
| **\r** | Retour au début de ligne | **\f** | Saut de page (imprimante) |
| **\0** | Définit la fin d'une chaîne de caractères | **\v** | Tabulateur vertical |

Une séquence d'échappement est un couple de symboles dont le premier est le signe d'échappement '\'. Au moment de la compilation, chaque séquence d'échappement est traduite en un caractère de contrôle dans le code de la machine comme indiqué dans le tableau ci-dessus.

### 3.2.7. Caractères précédés de " % "

Le symbole '%' suivi d'un certain caractère permet d'afficher la valeur des variables en différents formats.

%b : afficher en binaire, %c : afficher un caractère, %d : afficher en décimal

%o : afficher en octal, %s : afficher une chaîne de caractères, %u : afficher en décimal non signé, %x : afficher en hexadécimal

## 4. Le noyau C du langage C++

Le C++ en tant que langage orienté objet peut être considéré comme une extension du langage C. Le noyau C du langage C++ ajoute quelques spécificités supplémentaires au langage C, mais d'un autre côté un certain nombre d'incompatibilités existent entre les langages C ANSI et C++. Par rapport au langage C, le C++ dispose d'un certain nombre de spécificités, et qui ne sont pas axées sur la P.O.O telles que :

- Emplacement libre des déclarations des variables.
- Nouvelle forme de commentaire ( utilisant "//")
- Arguments par défaut dans la déclaration des fonctions.
- Sur-définition des fonctions.
- Fonction en ligne ('inline").
- Opérateurs "new" et "delete".

Dans les paragraphes suivants, on expliquera les quatre dernières spécifications du C++.

### 4.1. Arguments par défaut dans la déclaration des fonctions

L'exemple suivant montre, comment déclarer une fonction avec des arguments par défaut.

**Exemple 1.6:**

```
// Exemple sur les arguments par défauts
#include<iostream.h>
void main()
```

```
{
  int n=5,l=3,k=2;
  void fct1( int, int=0,int=8); //prototype de la fonction avec deux valeurs par défaut
  fct1(n,l,k); // appel normal de la fonction
  fct1(n,l); // appel deux arguments
  fc1(n); // appel avec un seul argument
}
void fct1(int a, int b, int c)
{
    cout<<"premier argument : "<<a<<;
    cout<<"\t deuxieme argument : "<<b<<;
    cout<<"\t troisieme argument : "<<c<<endl;
}
```

L'exécution du programme précédent donne le résultat suivant :

premier argument : 5 deuxieme argument : 3 troisieme argument : 2

premier argument : 5 deuxieme argument : 3 troisieme argument : 8

premier argument : 5 deuxieme argument : 0 troisieme argument : 8

La déclaration de fct1 est réalisée par le prototype : void fct1( int, int=0,int=8);

On note que la déclaration du deuxième et du troisième argument est sous la forme : int=0 et int=8. Celles-ci précisent au compilateur qu'en cas d'absence de ces arguments dans un éventuel appel de fct1(), ils vont prendre (deuxième et troisième arguments) par défaut les valeurs 0 et 8 respectivement dans la fonction fct1().

**Remarque:**

D'une manière générale, lorsqu'une déclaration prévoit des valeurs par défaut dans une fonction, les arguments concernés doivent obligatoirement être les derniers de la liste.

### 4.2. Sur-définition des fonctions

On parle de sur-définition des fonctions lorsqu'un même nom d'une fonction possède plusieurs significations différentes. Le choix de l'une des significations se faisant en fonction du contexte.

**Exemple 1.7:**

```
// Exemple sur les sur-définition des fonctions
```

```
#include<iostream.h>
void main()
{
   void fct(int);
   void fct(float);
   int n=5,p;
   double x=2.5;
   fct(n);
   fct(x);
}
void fct(int a)  // définition de la première fonction
{
  cout<<"valeur 1 est : "<<a<<"\n";
}
void fct(float a)  // définition de la deuxième fonction
{
  cout<<"valeur 2 est : "<<a<<"\n";
}
```

L'exécution du programme précédent donne le résultat suivant :

valeur 1 : a=5

valeur 2 : a=2.5

**4.3. Fonctions en ligne**

Lorsqu'une fonction est appelée lors de l'exécution d'un programme, celui-ci effectue un saut pour aller l'exécuter. A la fin de celle-ci, il revient à l'instruction juste suivant l'appel. Le programme effectuera autant de sauts que la le nombre d'appel de la fonction.

Lorsqu'une fonction est déclarée en ligne (précédée de mot "inline"), le compilateur effectue une copie de cette fonction à l'endroit de chaque appel, lors de la phase de compilation. Si la fonction est appelée dix fois dans un programme, alors dix copies seront crées à la fin de la phase de compilation. Lors de la phase d'exécution, le programme n'effectue aucun saut vers cette fonction. Ce qui permet de réduire le

temps d'exécution du programme. Toutefois, ceci entraîne une augmentation de l'espace mémoire occupé par le programme.

**Remarque:**

Généralement, une fonction est déclarée fonction en ligne, si elle vérifie simultanément les conditions :

- Elle contient un nombre très petit des instructions (de 3 à 5).
- Elle est appelée plusieurs fois dans le programme.

### 4.4. Opérateurs "delete" et "new"

En langage C, la gestion dynamique de mémoire fait appel à des fonctions de la bibliothèque standard telles que "maloc" et "free" qui sont utilisables aussi en C++. Mais, dans le contexte de la POO, C++ a introduit deux nouveaux opérateurs : "new" et "delete" adaptés à la gestion dynamique d'objets.

#### 4.4.1. Instruction "delete"

L'instruction "delete" permet de libérer un emplacement alloué préalablement par l'instruction "new".

**Syntaxe :**

delete adr;

#### 4.4.1. Instruction "new"

**Syntaxe:**

adr=new type_

| Déclaration | Instruction en C | Instruction en C++ | Description |
|---|---|---|---|
| int *adr; | adr=(int*)malloc(sizeof(int)); | adr=new int;<br>ou int *adr=new int; | Allouer un espace mémoire nécessaire pour un élément de type int. |
| char *adr; | adr=(char*)malloc(100); | adr=new char[100]; | Alloue un espace mémoire nécessaire pour un taleau de 100 caractères et place l'adress de début du tableau dans adr. |

Où type_ désigne un type quelconque. L'instruction "new" fournit comme résultat un pointeur (adr) pointant sur un emplacement d'objet de type "type_".

# Notions de base de la programmation en C++

## 1. Introduction

Pour l'écriture d'un programme en C++ que ce soit simple ou Compliqué, il est nécessaire de prévoir:

- Enchaîner deux traitements.
- Faire le choix entre plusieurs traitements.
- Répéter un traitement un certain nombre de fois.

## 2. Structures de contrôle

### 2.1. Branchements conditionnels

En informatique, il est nécessaire de pouvoir choisir entre deux ou plusieurs traitements. On trouve alors les instructions suivantes :

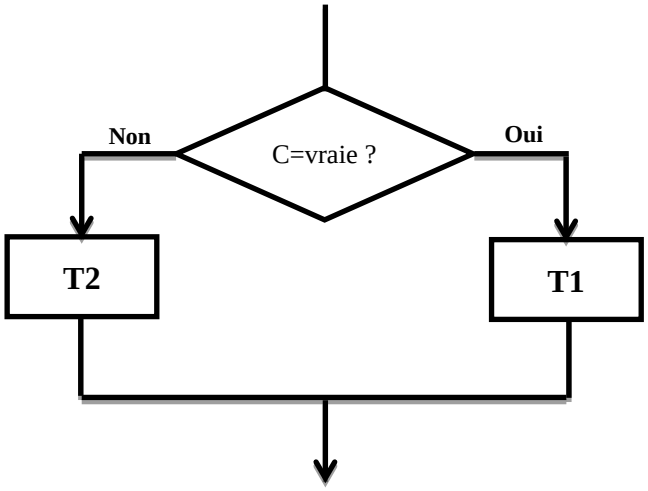
#### 2.1.1. Alternative complète

Il s'agit dans ce cas d'un choix binaire entre deux instructions (traitements). Autrement dit, si une condition « c » est vraie on exécute le traitement $T_1$, si non on exécute $T_2$.

**Syntaxe:**

| *Algorithme* | *En langage C++* |
|---|---|
| Si c vraie alors | if (c) |
| { | { |
| T1 ; | T1 ; |
| } | } |
| Si non | else |
| { | { |
| $T_2$ ; | $T_2$ ; |
| }Fsi | } |

Sous forme d'organigramme :




**Remarque:**

Le traitement $T_1/T_2$ peut être une ou plusieurs instructions. Dans le cas d'une seule instruction, on peut enlever les deux accolades.

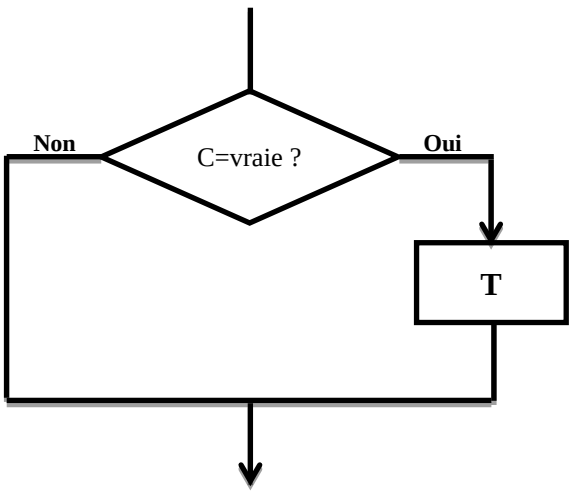
### 2.1.2. Alternative incomplète

Dans ce cas un traitement « T » est exécuté si et seulement si une condition « c » est vraie. Dans le cas contraire (la condition « c » est fausse), rien n'est à exécuter.

**Syntaxe:**

| *Algorithme* | *En langage C++* |
|---|---|
| Si « c » est vraie alors | if ( c) |
| { | { |
| T ; | T ; |
| } Fsi | } |

Sous forme d'organigramme :



**Exercice 2.1:**

Ecrire un programme C++ permettant de calculer le maximum de trois nombres A, B et C, puis faire l'affichage du résultat sur l'écran.

### 2.1.3. Branchement multiple (switch)

L'instruction « switch » permet de faire le choix entre plusieurs traitements.

**Syntaxe:**

| *Algorithme* | *En langage C++* |
|---|---|
| case choix | switch(choix) |
| Debut | { |
| expr 1 :instr1 ; | case 'expr1' :instr1 ; |
| expr 2 :instr2 ; | case 'expr2' :instr2 ; |
| ……..; | ……; |
| default: instr par défaut ; | default: instr par défaut ; |
| Fin | } |

**Exemple 2.1:**

```
#include <iostream.h>
#include<conio.h>
main()
{
int day ;
cout<<" Taper un numéro de jour \n"<<"Compris entre 1 et 3:" ;
day=getchar() ;
switch(day)
 {
case '1' :cout<<" jour 1 \n";
break;
case '2': cout<<"jour 2\n";
break;
case '3': cout<<"jour 3\n";
break;
default: ;
```

```
  }
return 0;
}
```

### 2.2. Branchements inconditionnels

#### 2.2.1. L'instruction « break »

Elle permet d'interrompre le déroulement d'une boucle, en passant à l'instruction qui suit cette boucle.

**Exemple 2.2:**

```
#include <iostream.h>
void main (void)
{
char n ;
cout<<"Donner un nombre entier positif  >= 2 " ;
cin>>n;
switch(n)
{
case '0':cout<<"le nombre introduit = nul \n" ;
break ;
case '1':cout<<"le nombre introduit = un \n";
break ;
case '2':cout<<"le nombre introduit = deux \n" ;
break ;
default :cout<<" le nombre introduit est supérieur à 2 \n" ;
 }
printf(" Au Revoir " ) ;
}
```

**Remarque:**

En l'absence de l'instruction « break », on continue l'exécution des instructions correspondant aux autres cas qui suivent le choix désiré.

### 2.2.2. L'instruction « continue »

L'instruction « continue » dans une boucle permet un branchement inconditionnel au début de cette boucle.

**Exemple 2.3:**

```
#include <iostream.h>
main( )
{
int n=10,i ;
for(i=0 ;i<=n ;i++)
{
if (i%4==0)
continue;
cout<<" Le Carré de"<< i<<" est="<<i*i ;
}
}
```

### 2.2.3. L'instruction « goto »

Elle permet un branchement inconditionnel vers un endroit quelconque du programme et qui est indiqué par une étiquette.

## 3. Structures itératives (les boucles)

### 3.1. Définition

Souvent, on a besoin de répéter la même action plusieurs fois. Au lieu d'écrire autant de fois qu'il faut les mêmes instructions, on fait appel à une boucle. On trouve trois types d'instructions :

### 3.1.1. L'instruction « while »

Cette instruction exécute une liste d'instructions tant qu'une condition « c » est réalisée.

**Syntaxe:**

```
while( c)
{
   Liste d'instructions ;
}
```

**Exemple 2.4:**

```
i=1;
while (i<=6)
{
cout<<i ;
i=i+1;
}
```

### 3.1.2. L'instruction « do … while »

L'instruction « do … while » permet d'exécuter d'abord les instructions de la boucle, puis elle teste la condition « c ».

**Syntaxe :**

```
do {
liste d'instructions ;
}while( c) ;
```

**Exercice 2.2:**

Ecrire un programme C++ qui permet de calculer l'expression :

$$Som = \sum_{i=1}^{N} (1+i).i$$

En utilisant :

- L'instruction « while »
- L'instruction « do … while »

**Solution:**

*1$^{er}$ cas : en utilisant « while »*

```
#include<iostream.h>
void main( )
{
inti,n,som ;
i=1 ;
som=0 ;
cout<<"Donner un entier positif :" ;
cin>>n ;
```

```
while(i<=n)
{
som=som+(1+i)*i;
i=i+1;
}
cout<<" pour n="<<n<< "la  somme est égale:"<<som;
}
```

*2ème cas: en utilisant « do…while »*

```
#include<iostream.h>
void main()
{
int i,n,som;
i=1;
som=0;
cout<<"donner un entier positif :";
cin>>n;
do {
som+=som+(1+i)*i ;
i++;
}while(i<=n);
cout<<"pour n="<<n<<"la somme est égale:"<<som;
}
```

### 3.1.3. L'instruction « for »

Cette instruction consiste à mettre en œuvre un certain nombre d'itérations, en utilisant une variable de contrôle.

**Syntaxe:**

```
for(initialisation_compteur ;condition_arrêt ;modification_compteur)
{
Liste d'instructions;
}
```

**Exemple 2.5:**

```
for(i=1 ;i<=6 ;i++)
{
cout<<i;
}
```

**Remarque:**

a. L'instruction i++ est équivalente à i=i+1. En langage C++, on peut avoir i++où ++i, les deux instructions incrémentent de 1 la valeur de i, mais leurs valeurs dans les expressions sont différentes.

**Exemple 2.6:**

i=5 ; x=i++ ; donne : $\begin{cases} x = i \\ i = i + 1 \end{cases} \rightarrow \begin{cases} x = 5 \\ i = 6 \end{cases}$

i=5 ; x=++i ; donne : $\begin{cases} i = i + 1 \\ x = i \end{cases} \rightarrow \begin{cases} i = 6 \\ x = 6 \end{cases}$

b. Les mêmes remarques sont applicables pour les opérations de décrémentation--i et i--.

c. L'instruction som+=i est équivalente à som=som+i.

d. La remarque précédente est applicable aussi pour les autres opérations arithmétiques.

**4. Les tableaux**

Un tableau est un ensemble d'éléments de même type désigné par un identificateur. Une donnée simple (scalaire) contient une seule information tandis qu'une donnée de type structurée contient une collection d'informations. Une donnée de type structuré est homogène (tableau) si toutes ses informations sont de même types. Dans le cas contraire, elle est hétérogène.

**4.1. Tableaux à une dimension**

Un tableau est une liste des valeurs repérées par un indice. Il est alors nécessaire de préciser d'une part la taille du tableau et d'autre part le type de ses éléments.

**Syntaxe:**

```
type nom_tab[taille] ;
```

L'instruction « type » correspond à l'indicateur du type des données à stocker dans le tableau. L'élément « nom_tab » est le nom du tableau à déclarer. L'élément « taille » définit le nombre d'éléments du tableau et doit être présenté entre deux crochets.

**Exemple 2.7:**

```
int t[3] ;// t est un tableau de trois entiers
float v[5] ;// v est un tableau de cinq nombres réels
char jour[7] ;// jour est un tableau de sept caractères
```

**Exemple 2.8:**

```
#include <conio.h>
#include <iostream.h>
#define n3 // n est constant et =3
void main()
{
int i,tab[n];// déclaration de l'élément tab comme une variable de type tableau
clrscr( );
//lecture des éléments d'un tableau 1D
for(i=0 ;i<n ;i++)
{
cout<<"tab["<<i<<"]=";
cin>>tab[i];
}
//affichage des éléments du tableau tab
for(i=0 ;i<n ;i++)
cout<<"tab["<<i<<"]= "<<tab[i]<<"\n";
getchar( );
}
```

**Exercice 2.3:**

Soit à créer un vecteur v en mémoire qui contient les notes de cinq étudiants et les afficher sur écran.

**Solution:**

```
// La taille du vecteur v=5
```

```
//Le type des données : réel
//La dimension de vecteur v=1
#include <iostream.h>
#include <conio.h>
void main( )
{
const int n=5;
float v[n];
int i;
//lecture des éléments du tableau
for(i=0 ;i<n ;i++)
cin>>v[i];
// affichage des éléments du tableau
for(i=0 ;i<n ;i++)
cout<<"v["<<i<<"]="<<v[i];
getchar();
}
```

### 4.2. Tableaux à deux dimensions

Un tableau 2D (matrice) est une liste des valeurs repérées par deux indices. Il est alors nécessaire de préciser d'une part la taille de la matrice et d'autre part le type de ses éléments.

**Syntaxe:**

```
Type nom_tab[taille_x][taille_y] ;
```

**Exemple 2.9:**

```
int t[2][3] ;// t est une matrice des entiers de 2 lignes et 3 colonnes
float v[5][2] ;// v est une matrice des réels de 5 lignes et 2 colonnes
v[0][0]=2 ;// affecter au premier élément de la matrice v la valeur 2
```

**Exercice 2.4:**

Ecrire un programme C++ permettant de calculer la somme de deux matrices de taille 3x5 ou le contenu de ces matrices sont des réels.

**Solution:**

```
#include<iostream.h>
#include<conio.h>
void main()
{
const int n=3,m=5;
float t1[n][m],t2[n][m],t3[n][m] ;
int i,j ;
clrscr() ;
// lecture des éléments du tableau t1
for(i=0 ;i<n ;i++)
{
for(j=0 ;j<m ;j++)
{
cout<<"t1["<<i<<"]["<<j<<"]=";
cin>>t1[i][j];
}
}
//* lecture des éléments du tableau t2
for(i=0 ;i<n ;i++)
{
for(j=0 ;j<m ;j++)
{
cout<<"t2["<<i<<"]["<<j<<"]=";
cin>>t2[i][j];
}
}
// calcul des éléments du tableau t3
for(i=0 ;i<n ;i++)
{
for(j=0 ;j<m ;j++)
t3[i][j]=t1[i][j]+t2[i][j] ;
```

```
}
// affichage des éléments du tableau t3
for(i=0 ;i<n ;i++)
{
for(j=0 ;j<m ;j++)
cout<<"t3["<<i<<"]["<<j<<"]="<<t3[i][j];
}
getchar() ;
}
```

**Exercice 2.5:**

Ecrire un programme C++ permettant de calculer le produit de deux matrices, puis il affiche le résultat sur écran. Sachant que les éléments des deux matrices sont réels.

**Remarques:**

a. On peut initialiser les tableaux au moment de leur déclaration.

**Exemple 2.10:**

```
int t[5]={1,32,10,-4,2} ;
float v[3]={2.32,0.1,-5.92} ;
int v[2][3]={{1,5,7},{8,6,2}} ;
```

b. Pour les tableaux à plus de deux dimensions, on procède de la même manière, en ajoutant les éléments de dimensionnement ou les indices nécessaires.

**5. Les structures en C++**

Une structure est un ensemble d'informations qui ne sont pas obligatoirement de même type. Dans ce cas, il devient nécessaire d'utiliser un mécanisme permettant de regrouper de façon cohérente un certain nombre de variables, et c'est ce qu'offre le concept de structure.

**Exemple 2.11:**

Soit une variable décrivant une personne contenant par exemple son nom et prénom (de type chaîne de caractères), sa date de naissance (de type entier), la taille (de type réel). D'où la structure est la suivante :

```
typedef struct person
  {
```

```
    char nom[32] ;
    char prenom[30] ;
    int jour, mois, annee ;
   float taille ;
} persont ;
```

Cette structure est constituée des champs (nom, prénom, jour, mois, annee, taille). Alors que persont est une variable de type structure (person). Grâce à cette variable, on peut déclarer des variables de type structure :

persont ali ;

On peut accéder aux champs de la structure ali, de la façon suivante :

ali.jour=12 ;

ali.mois=3 ;

strcpy(ali.nom, “mohamed”) ;

**Remarque:**

Il est possible de définir un tableau de personnes:

persont v[4] ;

Dans ce cas, on accède aux champs des personnes :

v[2].annee=2014 ;

strcpy(v[1].prenom,”Hakim”);

**6. Les fonctions**

Le langage C++ permet de diviser un programme en plusieurs parties nommées modules ou fonctions. Un programme devient fastidieux et difficile à manipuler dés qu'il dépasse un certain nombre des lignes. La programmation modulaire permet:

- De découper un programme en plusieurs parties et de regrouper dans un programme principal les instructions qui contrôlent les enchaînements.
- D'éviter des séquences d'instructions répétitives.

**6.1. Structure d'une fonction en C++**

Une fonction est un sous-programme comportant un en-tête qui précise le nom de la fonction avec ses arguments s'ils existent et leurs types et d'un corps dans lequel se trouve les déclarations et les instructions à exécuter en cas d'appel de cette fonction.

**Syntaxe:**

```
type nom_fonction(type arg1, type arg2, ..., type argN)
{
  Déclarations des variables;
  Instructions;
 }
```

L'appel de la fonction est obtenue par:

```
x=nom_fonction(val1, val2, ..., valN);
```

Où val1, val2, val3, ..., valN ont respectivement les mêmes types que celles de arg1, arg2, ..., argN. De plus, la variable x a le même type que la valeur retournée par la fonction.

Chaque fonction possède son prototype qui a le même format que l'en-tête de la fonction. Le prototype d'une fonction constitue en quelque sorte sa déclaration.

**Exemple 2.12:**

```
#include<iostream.h>
#include<conio.h>
void main()
  {
    int i,l;
    l=2;
    int power(int ,int );// prototype de la fonction power()
    for(i=0;i<10;i++)
    cout<<l<<" à la puissance "<<i<<" = "<<power(l,i);
    return 0;
}
int power(int x,int n)
{
int i,p;
p=1;
for(i=1;i<=n;i++)
p*=x; // equivalent à p=p*x;
return(p);
```

}

Ce programme calcul la puissance $n^{ième}$ de la valeur 2.

**Remarques:**

- L'exécution de tout programme C++ commence toujours par la fonction main().
- Une fonction qui ne retourne pas une valeur est une fonction de type void (une procédure en Pascal).

**Exemple 2.13:**

void message(int, float)

La fonction message() est une fonction de type void c'est-à-dire qu'elle ne retourne pas une valeur. cette fonction possède deux arguments de types entier et réel respectivement.

### 6.2. Arguments et variables locales

Dans l'exemple 2.12, les variables x et n sont des variables locales de type int qui n'ont de significations qu'au sein de la fonction power().Elles n'ont également aucun rapport avec d'éventuelles variables de mêmes noms qui pourraient êtres définies en dehors de la fonction power(). On dit que les variables x et n sont des arguments formels. Les variables *l* et i sont des variables effectives dont les valeurs sont effectivement transmises à la fonction power() lors de son appel. Les arguments de la fonction sont transmis par valeurs. La fonction appelée power() reçoit les valeurs de ces arguments par l'intermédiaire des variables formelles et non pas par leurs adresses.

### 6.3. Variables globales

On peut définir des variables globales, en dehors de toute fonction. Elles seront connues de toutes les fonctions qui seront compilées par la suite.

**Exemple 2.14:**

```
#include<iostream.h>
int i;// i est une variable globale
void main()
 {
   void fct(); // déclaration de la fonction fct()
   for(i=1;i<=2;i++)
     fct();
```

```
  }
void fct()
{
 cout<<"Bonjour Monsieur "<<i<<" fois\n";
}
```

Dans le programme ci-dessus, la variable i est déclarée comme une variable globale.

### 6.4. Les fonctions mathématiques en C++

Le langage C++ possède des fonctions mathématiques prédéfinies. Toutes les fonctions mathématiques de la bibliothèque C++ sont déclarées avec des prototypes dans un ou plusieurs fichiers en-tête (math.h, stdlib.h et complex.h)

**Exemple 2.15:**

| Fonction | Signification | Prototype | Fichier à inclure |
|---|---|---|---|
| abs() | Valeur absolue | int abs(int) | stdlib.h |
| acos() | Arc cosinus | double acos(double) | math.h |
| exp() | Exponentiel | double exp(double) | math.h |
| pow() | Puissance | double pow(double) | math.h |
| sqrt() | Racine carrée | double sqrt(double) | math.h, complex.h |

### 6.5. Les fonctions récursives

Une fonction peut s'appeler elle même. La récursivité peut prendre deux aspects:

***a. Récursivité directe :*** où une fonction comporte dans sa définition au moins un appel à elle même.

***b. Récursivité croisée :*** où l'appel d'une fonction entraîne celui d'une autre fonction qui à son tour appelle la fonction initiale.

**Exemple 2.16:** Calcul du factoriel d'un entier positif

```
#include<iostream.h>
void main()
{
 int i, fact(int);
```

```
 i=1;
while(i!=-1)
 {
   cout<<"donner un nombre positif:";
   cin>>i;
   if(i<0) break;
   cout<<i<<"!= "<<fct(i);
  }
printf("Au revoir");
}
int fact(int n)
{
  if(n>1) return(fact(n-1)*n);
else return(1);
}
```

**Exercice 2.7:**

Ecrire une fonction récursive calculant la valeur de la fonction d'ackermann A(m,n) définie:

$$A(m,n)=\begin{cases} A(m-1,A(m,n-1)) & pour\ m>0\ et\ n>0 \\ A(0,n)=n+1 & pour\ n>0 \\ A(m,0)=A(m-1,1) & pour\ m>0 \end{cases}$$

**Solution:**

```
#include<iostream.h>
void main()
{
  float acker(int, int);
int i,j;
float y;
cout<<"Donner deux nombres positifs:";
cin>>i>>j;
y=acker(i,j);
```

```
cout<<"acker("<<i<<","<<j<<")= "<<y;
}
float acker(int m, int n)
{
  if((m>0)&(n>0)) return(acker(m-1,acker(m,n-1));
else if((m==0)&(n>0)) return(n+1);
else if((n==0)&(m>0)) return(acker(m-1,1));
}
```

## 7. Les fichiers

En informatique, un fichier est un ensemble d'informations stockées sur un support, réuni sous un même nom et manipulé comme une unité. Le langage C++ offre la possibilité de lire et d'écrire des données dans un fichier. Pour des raisons d'efficacité, les accès à un fichier se font par l'intermédiaire d'une mémoire-tampon (*buffer*), ce qui permet de réduire le nombre d'accès aux périphériques (disque dur par exemple).

Pour pouvoir manipuler un fichier, un programme a besoin d'un certain nombre d'informations : l'adresse de l'endroit (buffer) où se trouve le fichier, la position de la tête de lecture, le mode d'accès au fichier (lecture ou écriture), etc. Ces informations sont rassemblées dans une structure de type "**FILE ***", qui est définie dans la bibliothèque "**iostream.h**". Un objet de type FILE * est appelé *flot de données* (en anglais, stream). Pour chaque opération sur le fichier, il faut l'ouvrir avant toute manipulation et le fermer à la fin.

### 7.1. Ouverture et fermeture des fichiers

#### 7.1.1. Fonction fopen

Cette fonction, de type FILE* ouvre un fichier et lui associe un flot de données. Sa syntaxe est :

fopen("*nom-de-fichier*","*mode*")

La valeur retournée par fopen est un flot de données. Si l'exécution de cette fonction ne se déroule pas normalement, la valeur retournée est le pointeur NULL. Il est donc recommandé de tester si la valeur renvoyée par la fonction fopen est égale à NULL afin de détecter les erreurs (lecture d'un fichier inexistant par exemple).

Le premier argument de fopen est le nom du fichier concerné, fourni sous forme d'une chaîne de caractères.

Le second argument, *mode*, est une chaîne de caractères qui spécifie le mode d'accès au fichier. Les spécificateurs de mode d'accès diffèrent suivant le type du fichier considéré. On distingue:

1. Les *fichiers textes*, pour lesquels les caractères de contrôle (retour à la ligne par exemple) seront interprétés en tant que tels lors de la lecture et de l'écriture.
2. Les *fichiers binaires*, pour lesquels les caractères de contrôle ne sont pas interprétés.

Les différents modes d'accès sont les suivants:

| **Mode** | **Description** |
|---|---|
| "r" | Ouverture d'un fichier texte en lecture |
| "w" | Ouverture d'un fichier texte en écriture |
| "a" | Ouverture d'un fichier texte en écriture à la fin |
| "rb" | Ouverture d'un fichier binaire en lecture |
| "wb" | Ouverture d'un fichier binaire en écriture |
| "ab" | Ouverture d'un fichier binaire en écriture à la fin |
| "r+" | Ouverture d'un fichier texte en lecture/écriture |
| "w+" | Ouverture d'un fichier texte en lecture/écriture |
| "a+" | Ouverture d'un fichier texte en lecture/écriture à la fin |
| "r+b" | Ouverture d'un fichier binaire en lecture/écriture |
| "w+b" | Ouverture d'un fichier binaire en lecture/écriture |
| "a+b" | Ouverture d'un fichier binaire en lecture/écriture à la fin |

Ces modes d'accès ont pour particularités :

- Si le mode contient la lettre r, le fichier doit exister.
- Si le mode contient la lettre w, le fichier peut ne pas exister. Dans ce cas, il sera cré. Si le fichier existe déjà, son ancien contenu sera perdu.
- Si le mode contient la lettre a, le fichier peut ne pas exister. Dans ce cas, il sera cré. Si le fichier existe déjà, les nouvelles données seront ajoutées à la fin du fichier précédent.

Trois flots standards peuvent être utilisés en C++ sans qu'il soit nécessaire de les ouvrir ou de les fermer:

- stdin (standard input) : unité d'entrée (par défaut, le clavier) ;
- stdout (standard output) : unité de sortie (par défaut, l'écran) ;
- stderr (standard error) : unité d'affichage des messages d'erreur (par défaut, l'écran).

### 7.1.2. Fonction fclose

Elle permet de fermer le flot qui a été associé à un fichier par la fonction fopen. Sa syntaxe est :

fclose(*flot*)

où *flot* est le flot de type FILE* retourné par la fonction fopen correspondant. La fonction fclose retourne un entier qui vaut zéro si l'opération s'est déroulée normalement (et une valeur non nulle en cas d'erreur).

### 7.1.3. Les entrées-sorties formatées

#### 7.1.3.1. Fonction d'écriture fprintf

La fonction fprintf(), analogue à celle de printf(), permet d'écrire des données dans un fichier. Sa syntaxe est:

fprintf(*flot*,"*chaîne de contrôle*",*expression_1, ..., expression_n*)

où *flot* est le flot de données retourné par la fonction fopen(). Les spécifications de format utilisées pour la fonction fprintf() sont les mêmes que pour printf().

#### 7.1.3.2. Fonction de saisie fscanf

La fonction fscanf(), analogue à celle de scanf(), permet de lire des données à partir d'un fichier. Sa syntaxe est semblable à celle de scanf :

fscanf(*flot*,"*chaîne de contrôle*",*argument_1,...,argument_n*)

où *flot* est le flot de données retourné par fopen. Les spécifications de format sont ici les mêmes que celles de la fonction scanf().

#### 7.1.3.3. Impression et lecture des caractères

Similaires aux fonctions getchar() et putchar(), les fonctions fgetc() et fputc() permettent respectivement de lire et d'écrire un caractère dans un fichier. La fonction fgetc(), de type int, retourne le caractère lu dans le fichier. Elle retourne la constante EOF lorsqu'elle détecte la fin du fichier. Son prototype est:

int fgetc(FILE* *flot*);

où *flot* est le flot de type FILE* retourné par la fonction fopen(). Comme pour la fonction getchar(), il est conseillé de déclarer de type int la variable destinée à recevoir la valeur de retour de fgetc() pour pouvoir détecter correctement la fin de fichier.

La fonction fputc() écrit des caractères dans le flot de données:

int fputc(int *caractere*, FILE **flot*)

Elle retourne l'entier correspondant au caractère lu (ou la constante EOF en cas d'erreur).

Il existe également deux versions optimisées des fonctions fgetc() et fputc() qui sont implémentées par des macros. Il s'agit respectivement de getc() et putc(). Leur syntaxe est similaire à celle de fgetc() et fputc() :

int getc(FILE* *flot*)

int putc(int *caractere*, FILE **flot*)

Ainsi, le programme suivant lit le contenu du fichier texte "entrée.txt", et le recopie caractère par caractère dans le fichier "sortie.txt" :

```
#include <iostream.h>
#include <stdlib.h>
#define ENTREE "entree.txt"
#define SORTIE "sortie.txt"
void main(void)
{
 FILE *f_in, *f_out;
 int c;

 if ((f_in = fopen(ENTREE,"r")) == NULL)
   {
    fprintf(stderr, "\nErreur: Impossible de lire le fichier %s\n",ENTREE);
    return(EXIT_FAILURE);
   }
 if ((f_out = fopen(SORTIE,"w")) == NULL)
   {
    fprintf(stderr, "\nErreur: Impossible d'écrire dans le fichier %s\n",SORTIE);
    return(EXIT_FAILURE);
   }
 while ((c = fgetc(f_in)) != EOF)
  fputc(c, f_out);
 fclose(f_in);
 fclose(f_out);
 return(EXIT_SUCCESS);
}
```

### 7.1.4. Les entrées-sorties binaires

Les fonctions d'entrées-sorties binaires permettent de transférer des données dans un fichier sans transcodage. Elles sont donc plus efficaces que les fonctions d'entrée-sortie standard, mais les fichiers produits ne sont pas portables puisque le codage des données dépend des machines.

Elles sont notamment utiles pour manipuler des données de grande taille ou ayant un type composé. Leurs prototypes sont :

```
size_t fread(void *pointeur, size_t taille, size_t nb, FILE *flot);
size_t fwrite(void *pointeur, size_t taille, size_t nb, FILE *flot);
```

où pointeur est l'adresse du début des données à transférer, taille est la taille des objets à transférer, et nb leur nombre. Rappelons que le type size_t, défini dans stddef.h, correspond au type du résultat de l'évaluation de sizeof. Il s'agit du plus grand type entier non signé.

## 8. Les pointeurs

Le langage C++ permet de manipuler des adresses où sont contenues les valeurs des variables, par l'intermédiaire des variables nommées pointeurs. On dispose de deux opérateurs:

- L'opérateur d'indirection * désigne le contenu de l'adresse qui le suit.
- L'opérateur d'adressage & fournit l'adresse de son opérande.

**Exemple 2.17:**

int *ad; // ad est un pointeur qui pointe vers un entier et il contient son adresse.

*ad: représente le contenu de l'adresse ad.

**Remarques:**

Le langage C++ autorise l'addition ou la soustraction d'un entier à un pointeur, et le résultat est de type pointeur. Il autorise aussi la différence de deux pointeurs de même type, et le résultat est un entier.

**Exemple 2.18:**

Soit le programme suivant:

```
#include<iostream.h>
void main()
{
```

```
 int a=10,b=20;
 void fct(int*,int*);
cout<<"Avant appel : a= "<<a<<"et b="<<b<<"\n";
fct(&a,&b);
cout<<"Après appel : a= "<<a<<"et b="<<b<<"\n";
}
fct(int *ad1,int *ad2)
{
 int x;
x=*ad1;
*ad1=*ad2;
*ad2=x;
}
```

**8.1. Relation entre tableaux et pointeurs**

En langage C++, l'identificateur d'un tableau désigne toujours l'adresse de début de ce tableau c'est-à-dire l'adresse du premier élément du tableau.

**Exemple 2.19:**

int t[10];

l'identificateur t est équivalent à &t[0]; t est donc un pointeur qui pointe vers le premier élément du tableau.

De même t+1 est équivalent à &t[1], ..., t+i est équivalent à &t[i]

Et t[i] est équivalent à *(t+i)

**Exemple 2.20:**

```
#include<iostream.h>
void main()
{
 int i;
float x[10],*p ;
p=x;
for(i=0;i<10;i++)  /* ou for(p=x;p<&x[10];p++) */
*(p+i)=1;
```

```
for(i=0;i<10;i++)
cout<<"élément "<<i<<" est : "<<*(p+i)<<"\n";
}
```

L'affectation p=x permet de pointer vers l'adresse du premier élément du tableau x.

**8.2. Pointeurs et fonctions**

Le langage C++ autorise la définition de variable destinée à pointer vers une fonction. En C++, le nom d'une fonction est traduit par le compilateur en adresse de cette fonction, ce qui ressemble à ce qui se passe avec les noms des tableaux.

**Exemple 2.21:**

```
int (*adf)( );/* déclaration d'une fonction par son adresse*/
adf=fct1;/* affectation de l'adresse de la fonction fct1() au pointeur adf */
```

**8.3. Pointeurs et fichiers :**

Soit l'exemple suivant permettant la création d'un fichier.

**Exemple 2.22:**

```
#include<iostream.h>
void main()
{
 File *fptr;
int i;
char c[];
clrscr();
fptr=fopen("c:\\toto.txt","w");
if(fptr==NULL)
cout<<"Erreur d'ouverture";
else
{
for(i=0;i<=5;i++)
{
gets(c);
fputs(c,fptr);
}
```

```
}
fclose(fptr);
}
```

**Exemple 2.23:**

Ce programme permet de lire un fichier

```
#include<iostream.h>
#include<string.h>
#include<stdlib.h>
void main()
{
File *fptr;
char c[3];
clrscr();
fptr=fopen("c:\\toto.txt","r");
if(fptr==NULL)
{
 cout<<"Erreur d'ouverture";
exit(1);
}
else
{
while(!eof(fptr))
{
 fgets(c,strlen(c)+1,fptr);
  cout<<c;
}
}
fclose(fptr);
getch();
}
```

L'écriture dans un fichier est réalisée par la fonction fwrite( ), dont la syntaxe est donnée par:

fwrite(&n, m, x, y);

&n: désigne l'adresse du bloc d'informations

m: est la taille d'un bloc en octets

x: est le nombre de blocs que l'on souhaite transférer

y: est l'adresse de la structure décrivant le fichier (sortie)

La lecture des données depuis un flux (fichier) est effectuée par la fonction fread( ), dont la syntaxe est donnée par:

fread(void *ptr, x, y, File *fptr);/* fread lit un certain nombre d'éléments des données de taille identique depuis un flux en entrée vers un bloc mémoire */

ptr: est un pointeur qui pointe sur un block recevant les données

x: est la longueur d'élément à lire en octets

n: est le nombre d'éléments à lire

fptr: est un pointeur sur un flux en entrée.

Le nombre d'octets lus est (x*y ).

**Exemple 2.24:**

```
#include<iostream.h>
void main()
{
  char nom_fich[20];
  int n;
  File *entree;
  cout<<"nom du fichier à lister :";
 cin>>nom_fich;
sortie=fopen(nom_fich,"rb");
while( )
{
  fread(&n,2,1,entree);
  if(!feof(entree))
  cout<<"\n"<<n;
```

```
 }
fclose(entree);
}
```

**Exemple 2.25:**

Ce programme permet d'enregistrer séquentiellement dans un fichier une suite de nombres entiers.

```
#include<iostream.h>
void main()
{
  char nom_fich[20];
  int n;
  File *sortie;
  cout<<"nom du fichier à créer:";
  cin>>nom_fich;
 sortie=fopen(nom_fich,"wb");
 do
   {
     cout<<"donner un entier: ";
     cin>>n;
     if(n)
        fwrite(&n,2,1,sortie);
   } while(n);
fclose(sortie);
}
```

**9. Référence:**

Une référence peut être considérée comme un surnom d'une variable. Ceci signifie que l'on peut modifier le contenu de la variable en utilisant sa référence.

Une référence ne peut être initialisée qu'une seule fois : à la déclaration. Elle ne peut référencer qu'une seule variable.

**Syntaxe:**

**type &** nom_ref=nom_var;

L'opérateur "**&**" est utilisé pour déclarer une référence sur une variable. "**nom_ref**" désigne une référence de la variable "**nom_var**" qui est de type "**type**".

Généralement, ont utilise la notion de référence pour les cas suivants :

- Référencer une variable simple (entier, réel, ...etc.).
- Référencer une variable de type objet.
- Arguments d'une fonction.

**Exemple 2.26:**

```
//Référencier une variable simple
#include<iostream.h>
void main(){
int n=5,&r=n;
cout<<"Avant traitements : n= "<<n<<"et r= "<<r<<"\n";
r++;
cout<<"Après traitement 1 : n= "<<n<<"et r= "<<r<<"\n";
n*=3;
cout<<"Après traitement 2 : n= "<<n<<"et r= "<<r;
}
```

L'exécution du programme précédent fournit le résultat suivant :

Avant traitement : n= 5 et r= 5

Après traitement 1 : n= 6 et r=6

Après traitement 1 : n= 18 et r=18

Donc, toute manipulation sur la référence r de la variable n est en réalité une manipulation de la variable n elle même.

En C++, on peut aussi référencer une variable de type objet.

**Exemple 2.27:**

```
//Référence sur un objet
#include<iostream.h>
class covid19{
int sous_t ,gueris,deces,;
public:
void situation();
```

```
void afficher(){
cout<<"Nombre des personnes sous traitement = "<< sous_t <<"\n";
cout<<"Nombre des personnes guéris = "<< gueris<<"\n";
cout<<"Nombre des personnes décès = "<< deces;
}
};
void covid19::situation(){
cout<<"\n Donner la situation de Covid19 : (Guéris, Sous traitement, Décès)";
cin>> sous_t>> gueris>>deces;
}
void main(){
Covid19 virus,&ref=virus;// ref est une référence de l'objet virus
ref. situation();
ref.afficher();
}
```

### 9.1. Pointeurs et références

Les pointeurs et les références fonctionnent presque de la même façon. Il existe cependant des situations dans lesquelles l'utilisation des pointeurs s'impose :

- Un même pointeur est utilisé pour pointer différents objets, car une référence ne peut référencer qu'un seul objet à la fois.
- S'il existe une possibilité que l'objet pointé soit nul.

## 10. Allocation mémoire

Une liste linéaire (un tableau) se caractérise par le fait que les données sont stockées d'une façon séquentielle sur un support mémoire (mémoire centrale où secondaire). Il existe deux types d'allocations mémoires :

### 10.1. Allocation fixe

La dimension de la liste (tableau) est fixée par une déclaration au moment de la compilation.

### 10.2. Allocation dynamique

Permet de définir des tailles variables des tableaux.

### 10.3. Les plies

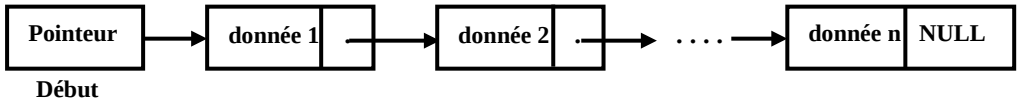
Il s'agit d'une structure linéaire dans laquelle, les insertions et les effacements ont lieu à une seule des extrémités de la liste (le haut de la pile). Donc, il s'agit d'une structure LIFO (Last In First Out).

### 10.4. Files d'attentes (Queues)

Il s'agit d'une structure linéaire FIFO (First In First Out).

### 10.5. Listes chaînées

Au lieu d'utiliser une allocation séquentielle de la mémoire, on peut recourir à une allocation plus flexible qui sert à associer à chaque donnée un pointeur vers l'élément suivant. Ce type d'allocation s'appelle aussi allocation chaînée.




**Exemple 2.28:**

```
#include<alloc.h>
#define Npmax 20
typedef struct{
int num;
float x;
float y;
}spoint;
void main()
{
 spoint *adr[Npmax];
int num,rang;
float x,y;
while(printf("numero,x,y:"),scanf("%d %f %f",&num,&x,&y)
{
  adr[rang]=(spoint*)malloc(sizeof(spoint));
adr[rang]->num=num;
adr[rang]->x=x;
adr[rang]->y=y;
```

```
rang++;
}
}
```

**Exemple 2.29:**

```
#include<alloc.h>
typedef struct el{
int num;
float x,y;
struct el *suit;
}s_point;
void main()
{
creation(s_point **adeb);
s_point *deb;
creation(&deb);
}
/* déclaration de la fonction*/
creation(s_point **deb)
{
  int num;
  float x,y;
  s_point *courant;
*deb=0;
while(printf("numero,x,y:"),scanf("%d %f %f",&num,&x,&y),num)
{
 courant=(s_point*)malloc(sizeof(s_point));
courant->num=num;
courant->x=x;
courant->y=y;
courant->suit=*adeb;;
*adeb=courant;
```

```
}
}
```

# Classes et objets

## 1. Introduction

La caractéristique la plus importante de C ++ est la notion de "classe". Son importance est soulignée par le fait que **Stroustrup** a initialement donné le nom "C avec classes" au langage C++. Une classe est une extension de l'idée de structure utilisée en C. C'est une nouvelle façon de créer et d'implémenter un type des données définies par l'utilisateur. Nous discuterons, dans ce chapitre, le concept de classe et les façons dont les classes peuvent êtres conçues, implémentées et appliquées.

## 2. Limites de la structure en C

Une importante limitation des structures en C réside dans le fait qu'elles ne permettent pas de masquer les données. Les membres de la structure peuvent êtres directement accessibles par n'importe quelle fonction et n'importe où dans leur champ d'application. En d'autres termes, les membres de la structure sont des membres publics. De plus, une structure ne contient que les données dans leur champs, et les fonctions qui accèdent à ces données arrivent de l'extérieur.

## 3. Extensions des structures

Le C++ prend en charge toutes les caractéristiques des structures définies en C. En outre, il a encore étendu ses capacités pour s'adapter à sa philosophie de la POO. Il tente de rapprocher le plus possible les types définis par l'utilisateur des types des données intégrés (int, float, etc.) et fournit également une fonction permettant de masquer les données. Le C++ intègre toutes ces extensions dans un autre type défini par l'utilisateur, appelé "**classe**". La plupart des programmeurs C++ ont tendance à utiliser les structures pour ne contenir que des données, et les classes pour gérer à la fois les données et les fonctions.

## 4. Définition et déclaration de classe

Une classe est une sorte de moule à partir duquel sont crées les objets qui s'appellent des instances de la classe considérée. Elle regroupe un ensemble des données (qui peuvent êtres des variables primitives ou des objets) et un groupe des méthodes de traitement de ces données. Donc, une classe est constituée de :

- Variables appelées **attributs** (on parle aussi des **variables membres** ou **membres de données**)
- Fonctions appelées **Méthodes (**ou **fonctions membres) :** décrivent les opérations qui seront appliquées aux instances de la classe.

**Syntaxe:**

```
class nom_de_la_classe
{
  // variables membres
   type variable_1;
   type variable_2;
   .....;
   type variable_N;
 // fonctions membres
   type fonction_1( );
   type fonction_2( );
   .....;
   type fonction_N( );
};
nom_de_classe c;
```

La variable **c** est déclarée comme étant un objet de type **nom_de_classe**.

**Exemple 3.1:**

```
class etudiant{
char nom[15],prenom[20];
float moyenne;
void initialiser();
};
etudiant v;
```

v est un objet de type "etudiant" contenant 3 variables membres dont deux variables de type chaînes de caractères, une variable de type réelle et une fonction membre initialiser() de type void.

### 4.1. Spécifications d'une classe

Une classe est un moyen de masquer les données et les fonctions qui leur sont associées d'une utilisation externe. En définissant une classe, nous créons un nouveau type des données abstraites qui peut être traité comme n'importe quel autre type des données intégrées. En général, la spécification d'une classe comporte deux parties :

- Déclaration de classe.
- Définition des fonctions membres de la classe.

La déclaration de classe décrit le type et la situation de ses membres, alors que les fonctions membres décrivent la manière dont ces fonctions sont mises en œuvres. La forme générale d'une déclaration de classe est :

```
class nom_classe
{
  private:
      déclaration des variables;
      déclaration des fonctions;
  public:
      déclaration des variables;
      déclaration des fonctions;
};
```

La déclaration de classe est similaire à celle de la structure. Le mot-clé "**class**" précise que ce qui suit est une donnée abstraite de type "**nom_classe**". Le corps d'une classe est entouré d'accolades et terminé par un point-virgule, et il contient la déclaration des variables et des fonctions qui sont collectivement appelées membres de la classe. Ces dernières sont généralement regroupées en deux sections, à savoir privée et publique. Les mots clés "**private**" et "**public**" sont connus sous le nom "**étiquettes de visibilité**". Les membres du groupe qui ont été déclarés comme étant privés ne sont accessibles qu'à l'intérieur du groupe. En revanche, les membres publics sont également accessibles depuis l'extérieur de la classe. Les membres d'une classe sont privés par défaut (si les deux étiquettes sont absentes).

En outre, seules les fonctions membres peuvent avoir accès aux données et aux fonctions membres privées de la même classe. Toutefois, les membres publics (fonctions et données) sont accessibles depuis l'extérieur de la classe. Ceci est illustré dans la figure.3.1. La liaison des données et des fonctions en une seule variable de type classe est appelée "**encapsulation**".

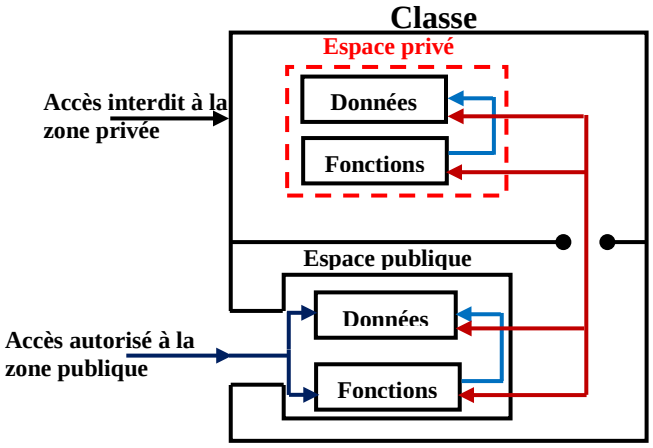



**Figure 3.1** : Accès aux membres du classe.

**Exemple 3.2:**

Soit la déclaration de la classe "item" suivante :

```
class item
{
  int a; //déclaration des variables membres
  float b; // a et b privées par défaut
public:
void getdata(int, float); //déclaration des fonctions membres
void put data(void);
};
```

Nous donnons généralement à une classe un nom significatif qui sera utilisé pour déclarer des instances de ce type de classe (objets). Les données membres sont privées par défaut tandis que les deux fonctions membres sont publiques par déclaration. La fonction getdata() peut être utilisée pour attribuer des valeurs aux variables membres "a" et "b", et putdata() pour afficher leurs valeurs. Ces fonctions fournissent le seul accès aux données membres de la classe.

```
item x; // x est objet de classe item
```

En C++, les variables de classe sont appelées "**objets**". Par conséquent, x est appelé un objet de type item.

L'espace mémoire nécessaire est alloué à l'objet au moment de sa déclaration. La spécification de classe, comme celle de structure, ne fournit qu'un modèle et ne crée aucun espace mémoire pour les objets.

Les objets peuvent également êtres crées lorsqu'une classe est définie en plaçant leur noms immédiatement après l'accolade de fermeture, comme nous le faisons dans le cas des structures.

```
class item
{
   ......;
   .......;
} x,y,z;
```

Les variables x, y et z sont déclarées comme des objets de type item. Cette pratique est rarement suivie car nous souhaitons déclarer les objets à proximité du lieu où ils sont utilisés et non au moment de la définition de la classe.

### 4.2. Accès aux membres de la classe et encapsulation

#### 4.2.1. Droits d'accès

Comme indiqué précédemment, les données privées d'une classe ne sont accessibles que par le biais des fonctions membres publiques de cette classe. La fonction main() ne peut pas contenir des déclarations qui accèdent directement aux variables "a" et "b" de l'exemple précédent. L'appelle des fonctions membres d'un objet s'effectue de la façon suivante :

```
nom_objet.nom_fonction(arguments);
```

Par exemple, l'appel de fonction getdata( ) :

```
x.getdata(10,3.5);
```

attribue la valeur 10 à la variable "a" et la valeur 3.5 à la variable "b" de l'objet x en mettant en œuvre la fonction getdata().

De même, l'instruction:

```
x.putdatad();
```

permet par exemple, d'afficher les valeurs des variables membres.

### 4.2.2. Encapsulation

Est le processus qui permet de grouper les membres de données et celles des fonctions dans une seule entité appelée classe. L'accès aux données est obtenu via les fonctions de la même classe, afin de les protéger. L'encapsulation est l'une des fonctionnalités populaires de la programmation orientée objet (POO).

L'encapsulation dans une classe est réalisée pou rendre :

- Tous les membres de données privés.
- Les fonctions membres qui opèrent sur ces données en mode de visibilité public.

### 4.3. Définition des fonctions membres (méthodes)

Une fonction membre d'une classe constitue une interface publique aux données privées de la même classe. Les fonctions membres ont des caractéristiques particulières qui sont souvent utilisées dans le développement des programmes. Ces caractéristiques sont les suivantes :

- Plusieurs classes différentes peuvent utiliser le même nom de fonction. L'étiquette d'appartenance résoudra leur portée.
- Les fonctions membres peuvent accéder aux données privées de la classe. Une fonction non membre ne peut pas le faire. (Toutefois, une exception à cette règle est la **fonction amie**).
- Une fonction membre appelle directement une autre fonction membre de même classe, sans utiliser l'opérateur "**.**".

En outre, une fonction membre peut être définie de deux façons différentes :

- En dehors de la déclaration de classe.
- A l'intérieur de la déclaration de classe.

### 4.3.1. En dehors de la déclaration de classe

Les fonctions membres qui sont déclarées à l'intérieur d'une classe doivent être définies à l'extérieur de la classe. Leur définition est très proche de celle des fonctions ordinaires. Elles doivent avoir un en-tête et un corps. Une différence importante entre une fonction membre et une fonction ordinaire est qu'une fonction membre intègre une "**étiquette d'identité**" d'appartenance à la classe dans son en-tête. Cette étiquette

indique au compilateur à quelle classe la fonction appartient. La forme générale de la définition d'une fonction membre est la suivante :

```
    type nom_classe :: nom_fonction(déclaration arguments)
{
  corps de la fonction;
}
```

L'étiquette d'appartenance nom_classe :: indique au compilateur que la fonction nom_fonction appartient à la classe "**nom_classe**". C'est-à-dire que la portée de la fonction est limitée au nom de classe spécifiée dans la ligne d'en-tête. Le symbole "**::**" est appelé l'opérateur de résolution de la portée.

Par exemple, considérons les fonctions membres précédentes getdata() et putdata(). Elles peuvent être définies comme suit :

```
void item :: getdata(int c, float d)
{
   a=c;
   b=d;
}
void item :: putdata(void)
{
  cout<<"a= "<<a<<"\n";
  cout<<"b= "<<b<<"\n";
}
```

### 4.3.2. A l'intérieur de la déclaration de la classe

Une autre manière de définir une fonction membre consiste à remplacer la déclaration de fonction par sa définition au sein de la classe. Par exemple, nous pouvons définir la fonction putdata( ) à l'intérieur de la classe comme suit :

```
class item
{
    int a;
    int b;
   public:
```

```
  void getdata(int c, float d);
  void putdata(void) // declaration inline function
    {
        cout<<"a="<<a<<"\n";
        cout<<"b="<<b<<"\n";
    }
};
```

Lorsqu'une fonction est définie à l'intérieur d'une classe, elle est traitée comme une fonction en ligne. Par conséquent, seules les petites fonctions sont définies à l'intérieur de la classe.

**4.4. L'imbrication des fonctions membres**

Une fonction membre peut être appelée en utilisant son nom à l'intérieur d'une autre fonction membre de la même classe. C'est ce qu'on appelle "**l'imbrication des fonctions membres**". Le programme suivant illustre cette caractéristique.

```
#include<iostream>
using namespace std
class set
{
   int m,n;
  public:
   void input(void);
   void display(void);
   int largest(void);
 };
int set :: largest(void)
{
   if(m>n)  return m;
   else return n;
}
void set :: input(void)
{
```

```
   cout<<"Enter les valeurs de m et n : \n";
  cin>>m>>n;
}
void set :: display(void)
{
cout<<"La valeur la plus grande est : "<<largest()<<"\n"; // appel de fonction membre
}
void main()
{
  set A;
  A.input();
 A.display();
}
```

**4.5. Fonctions amies**

Nous avons insisté tout au long de ce chapitre sur le fait que les membres privés ne sont pas accessibles depuis l'extérieur de la classe. C'est-à-dire qu'une fonction non membre ne peut pas avoir accès aux données privées d'une classe. Cependant, il pourrait y avoir une situation où nous souhaiterions que deux classes partagent une fonction particulière. Dans ce cas, le C++ permet de rendre la fonction commune compatible avec les deux classes, ce qui permet à la fonction d'avoir accès aux données privées de ces classes. Il n'est pas nécessaire qu'une telle fonction soit membre de l'une de ces classes.

Dans l'exemple suivant, la fonction xyz() est déclarée comme une fonction amie et par conséquent, elle peut manipuler les membres privés de la classe abc.

```
class abc
{
   ....
 public:
  .......
  friend void xyz();
};
```

### 4.6. Méthodes (fonctions membres) constantes

Une fonction membres est dite constante si elle ne modifie aucune donnée dans la classe. Elle est déclarée dans la classe comme suit :

```
void nom_fonction(int,int) const;
```

Le compilateur génère un message d'erreur si cette fonction tente de modifier les valeurs des données.

## 5. Classes et pointeurs

### 5.1. Pointeurs des membres de classe

Il est possible de prendre l'adresse d'un membre d'une classe et de l'attribuer à un pointeur. L'adresse d'un membre peut être obtenue en appliquant l'opérateur **&** à un membre de la classe. Par exemple, étant donné la classe suivante:

```
class A
{
private:
      int m;
public:
   void show();
};
```

Nous pouvons définir un pointeur vers les membres de classe comme suit :

```
A obj,*ip;
ip=&obj;
cout<<"m="<<ip->m<<endl; //le pointeur ip pointe vers le membre de donnée m pour afficher sa valeur
ip->show();//le pointeur ip pointe vers la fonction membre show() afin de l'exécuter.
```

## 6. Constructeur et destructeur d'une classe

### 6.1. Constructeur

Le constructeur d'une classe est une fonction membre de la classe ayant le même nom que celle-ci, et dont le rôle est de créer et d'initialiser les objets de cette classe. Il peut recevoir des paramètres mais ne retourne jamais de valeur. De plus, il ne possède pas de type. Le constructeur est automatiquement appelé lorsque l'objet est déclaré.

**Syntaxe:**

```
class nom_classe{
//déclaration des variables membres
.....
public:
nom_classe(arguments);//constructeur
.....
};
```

**Exemple 3.3:**

```
class classe{
                int x;
                char y;
                public:
                classe(int a, char b)//constructeur
                  { x=a;
                    y=b;
                   }
                void afficher()
                  {
                    cout<<"\n x="<<x<<"et y="<<y;
                   }
              };
```

**6.2. Destructeur**

A chaque fois qu'on déclare un constructeur, on doit déclarer un destructeur. Le destructeur permet de libérer l'espace mémoire alloué aux objets une fois leur utilisation terminée. Le destructeur possède également le même nom que la classe précédé du symbole "~" (tilde). C'est une fonction membre de la classe qui n'a pas de type.

**Syntaxe:**

```
class nom_classe{
//déclaration des variables membres
```

.....

public:

nom_classe(arguments);//constructeur

~nom_classe();//destructeur

.....};

# Héritage et polymorphisme

## 1. Introduction

La réutilisabilité est une autre caractéristique importante de la POO. Il est toujours agréable de pouvoir réutiliser quelque chose qui existe déjà plutôt que d'essayer de créer à nouveau la même chose. Cela permettrait non seulement de gagner du temps, mais aussi de réduire la confusion et d'accroître la fiabilité. Par exemple, la réutilisation d'une classe qui a déjà été testée, déboguée et utilisée plusieurs fois peut nous épargner l'effort de la développer et de la tester à nouveau.

Heureusement, le C++ favorise fortement le concept de réutilisabilité. Les classes C++ peuvent être réutilisées de plusieurs façons. Une fois qu'une classe a été écrite et testée, elle peut être adaptée par d'autres programmeurs pour répondre à leurs besoins. Cela se fait essentiellement en créant des nouvelles sous-classes, en réutilisant les propriétés des classes existantes. Le mécanisme de dérivation d'une nouvelle classe à partir d'une ancienne est appelé **héritage**. L'ancienne classe est appelée **classe de base** et la nouvelle classe est appelée **classe dérivée** ou sous-classe.

Le mécanisme qui permet à une classe dérivée d'hériter d'une partie ou de la totalité des caractéristiques d'une classe de base est appelé **héritage simple**. D'autre part, une classe peut également hériter des propriétés de plusieurs classes (**héritage multiple**) ou de plusieurs niveaux (**héritage multi-niveaux**). La figure 4.1 montre les différentes formes d'héritage qui pourraient être utilisées pour écrire des programmes extensibles.

## 2. Définition d'héritage

En plus de ses propres attributs, une classe dérivée peut être crée en spécifiant sa relation avec la classe de base. La syntaxe de déclaration d'une classe dérivée est :

```
class nom_ classe_derivee : mode_visibilité nom_classe_de_base
{
   .....
   ..... // membres de la classe dérivée
};
```

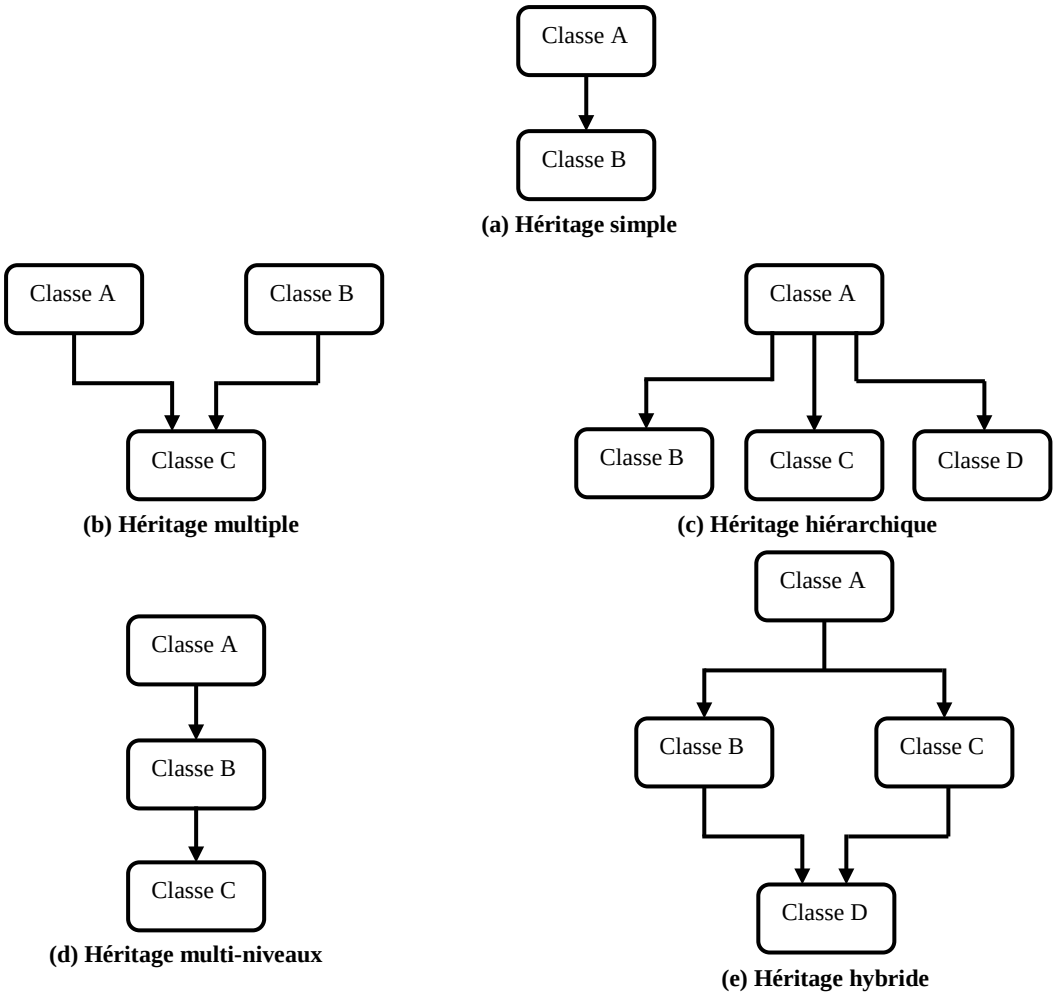



**Figure 4.1** : Différentes formes d'héritage.

Le mode de visibilité (mode_visibilité) spécifie si les caractéristiques de la classe de base sont dérivées de façon privées ou publiques. Elles sont par défaut privées.

Lorsqu'une classe de base est héritée à titre privé, les "membres publics" de la classe de base deviennent des "membres privés" dans la classe dérivée et, par conséquent, les membres privés de la classe de base ne sont accessibles que par les fonctions membres de la classe de base. D'autre part, lorsque la classe de base est héritée publiquement, les "membres publics" de la classe de base deviennent des "membres publics" de la classe dérivée et sont donc accessibles aux objets de la classe dérivée. Dans les deux cas, les membres privés ne sont pas hérités et, par conséquent, les membres privés d'une classe de base ne deviendront jamais des membres de sa classe dérivée.

En outre, nous pouvons ajouter nos propres fonctions et données membres à la classe dérivée, afin d'étendre les fonctionnalités de la classe de base.

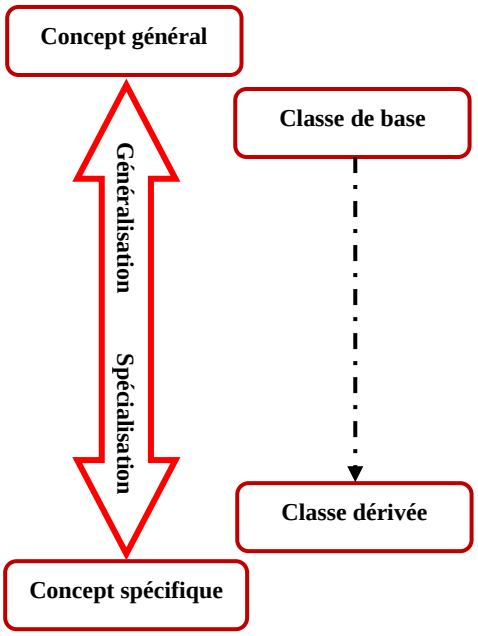



**Figure 4.2** : Concept d'héritage.

### 3. Règles d'héritage

Considérons un exemple simple pour illustrer l'héritage. Le programme ci-dessous montre une classe de base B et une classe dérivée D. La classe B contient un numéro de donnée privé, un membre de donnée public et trois fonctions membres publics. La classe D contient un membre de donnée privé et deux fonctions membres publics.

***a. Cas d'héritage public***

```
#include<iostream.h>
class B
{
  int a; // donnée privée, non héritable.
 public:
  int b; // donnée publique héritable.
 void get_ab();
 int get_a(void);
void show_a(void);
};
class D:public B  //  dérivation publique
{
```

```
  int c;
public:
void mul(void);
void display(void);
};
//-----------------------------------------------------------------------------------------------
void B::get_ab(void)
{
 a=5;b=10;
}
int B::get_a()
{
return a;
}
void B::show_a()
{
cout<<"a="<<a<<"\n";
}
void D::mul()
{
c=b*get_a();
}
void D::display()
{
cout<<"a= "<<get_a()<<"\n";
cout<<"b= "<<b<<"\n";
cout<<"c= "<<c<<"\n\n";
}
//-----------------------------------------------------------------------------------------------
---
int main()
```

```
{
 D d;
d.get_ab();
d.mul();
d.show_a();
d.display();
cou<<"=========================\n";
d.b=20;
d.mul();
d.display();
return 0;
}
```

Le résultat du programme est présenté ci-dessous :

```
a=5
a=5
b=10
c=50
=========================
a=5
b=20
c=100
```

La classe D est une dérivation publique de la classe de base B. Par conséquent, D hérite de la classe B tous les membres publics et conserve leur visibilité. Ainsi, un membre public de la classe de base B est également un membre public de la classe dérivée D. Les membres privés de B ne peuvent pas être hérités par D. La classe D, en effet, aura plus de membres que ce qu'elle contient au moment de la déclaration, comme le montre la figure 4.3.

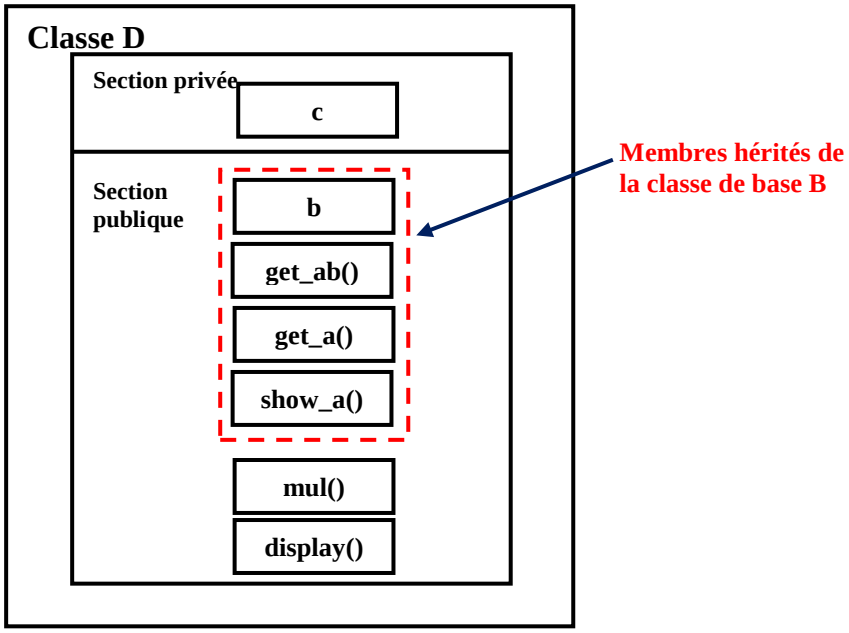



**Figure 4.3** : Construction de la classe dérivée D à partir d'un héritage public de la classe B.

Le programme illustre que les objets de la classe D ont accès à tous les membres publics de B. Bien que le membre de donnée "a" est privé dans la classe de base "B" et ne puisse pas être hérité, tandis que les objets de la classe dérivée "D" peuvent y accéder par le biais d'une fonction membre héritée de "B" (get_a() par exemple).

***b. Cas d'héritage privé***

Examinons maintenant le cas de la dérivation privée :

```
class D : private B  // dérivation privée
{
  int c;
public:
void mul(void);
void display(void);
};
```

Dans la dérivation privée, les membres publics de la classe de base deviennent des membres privés dans la classe dérivée. Par conséquent, les objets de la classe D ne peuvent pas avoir un accès direct aux fonctions membres publiques de la classe de base.

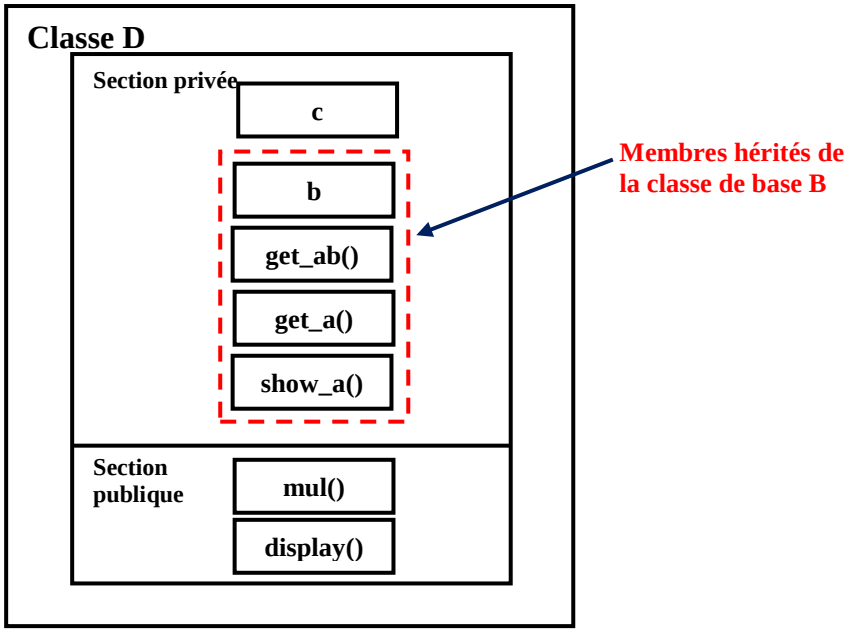



**Figure 4.4**: Construction de la classe dérivée D à partir d'un héritage privé de la classe B.

Les déclarations telles que :

d.get_ab(); // get_ab() est privée

d.get_a(); // get_a() est privée

d.show_a(); // show_a() est privée

ne fonctionneront pas. Cependant, ces fonctions peuvent être utilisées à l'intérieur de mul() et display() comme des fonctions ordinaires. Le programme suivant illustre ce principe.

```
#include<iostream.h>
class B
{
  int a; // private; not inheritable
 public:
  int b; // public ready for inheritance
 void get_ab();
 int get_a(void);
void show_a(void);
};
```

```
class D :  private B  // private derivation
{
  int c;
public:
void mul(void);
void display(void);
};
//-------------------------------------------------------------------------------------------
void B :: get_ab(void)
{
 cout<<"Entrer les valeurs de a et b :";
cin>>a>>b;
}
int B :: get_a()
{
return a;
}
void B :: show_a()
{
cout<<"a="<<a<<"\n";
}
void D :: mul()
{
  get_ab();
  c=b*get_a();
}
void D :: display()
{
  show_a();
 cout<<"b= "<<b<<"\n";
cout<<"c= "<<c<<"\n\n";
```

```
}
//-----------------------------------------------------------------------------------------------
int main()
{
 D d;
d.mul();
d.display();
d.mul();
d.display()
return 0;
}
```

### 3.1. Visibilité

Pour le type d'accès aux membres (attributs et méthodes), il faut maintenant tenir compte de la situation d'héritage comme indiqué dans le tableau ci-dessous :

- public (public) :
- privé (private) : les membres privés d'une classe ne sont accessibles que par les objets de cette classe et non par ceux d'une autre classe même dérivée.
- protégé (protected) : les membres protégés d'une classe ne sont accessibles que par les objets de cette classe et par ceux d'une classe dérivée.

| **Mode de dérivation** | **Statut dans la classe de base** | **Statut dans la classe dérivée** |
|---|---|---|
| Public | Public | Public |
| | protected | Protected |
| | private | Inacessible |
| Protected | Public | Protected |
| | Protected | Protected |
| | Private | inaccessible |
| Private | Public | Private |
| | Protected | Private |
| | Private | inaccessible |

### 3.2. Chaînage des constructeurs

Lorsqu'on utilise les constructeurs et les destructeur dans une classe de base et sa dans sa classe dérivée, le constructeur de la classe de base sera appelé en premier. Par contre, le destructeur de la classe dérivée sera appelé en premier lors de la destruction de l'objet.

## 4. Surchage des opératurs en C++

La surcharge des opérateurs en C++ permet de donner aux opérateurs une signification particulière pour un type de données. L'avantage de surcharge des opérateurs est d'effectuer différentes opérations sur le même opérande.

Pour surcharger un opérateur, une fonction d'opérateur spéciale est définie dans la classe comme suit :

```
class A
{
   .............
   public
       type operator symbol (arguments)
      {
          ...................
      }
   ..................
};
```

operator : est un mot clé du langage C++.

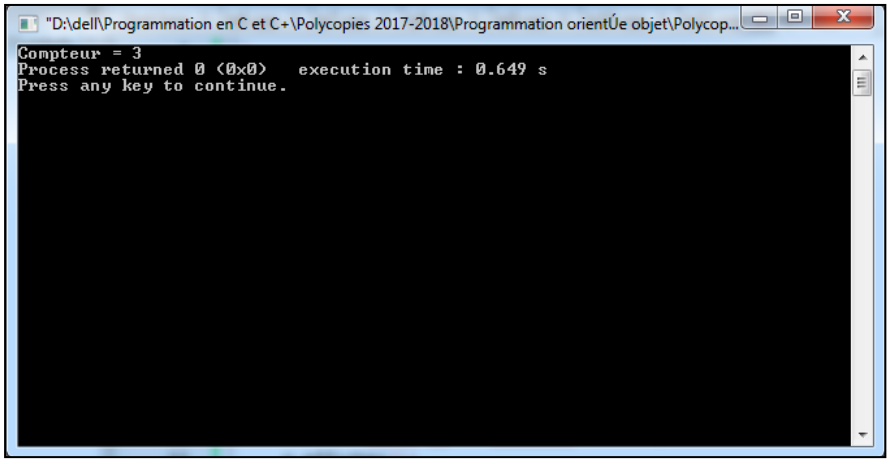
symbol désigne le symbole de l'opérateur que nous souhaitons surcharger (par exemple : +, <, -, ++, etc).

L'exemple suivant permet de surcharger l'opérateur "+" dans une classe afin d'incrémenter une valeur entière.

```
#include <iostream>
using namespace std;
class Count
{
  private:
```

```
    int i;
  public:
     Count(int n){
     i = n;
    }
     void operator +() {
       i = i+1;
     }
     void afficher() {
        cout << "Compteur = " << i;
     }
};
int main()
{
   Count c(2);
   +c;  // appel de la fonction "void operator +()"
   c.afficher();
   return 0;
}
```

Le résultat du programme est :



## 5. Surcharge et redéfinition de fonctions

La surcharge et la redéfinition de fonctions peuvent êtres considérée comme des exemples de polymorphisme, mais elles sont complètement différentes.

### 5.1. Surcharge de fonctions

La surcharge de fonctions produit plusieurs fonctions portant le même nom avec des paramètres différents.

**Exemple 4.1:**

```
float soustraction(int, int);
float soustraction (float, float);
float soustraction(int, float);
float soustraction (float, int);
```

### 5.2. Redéfinition de fonctions

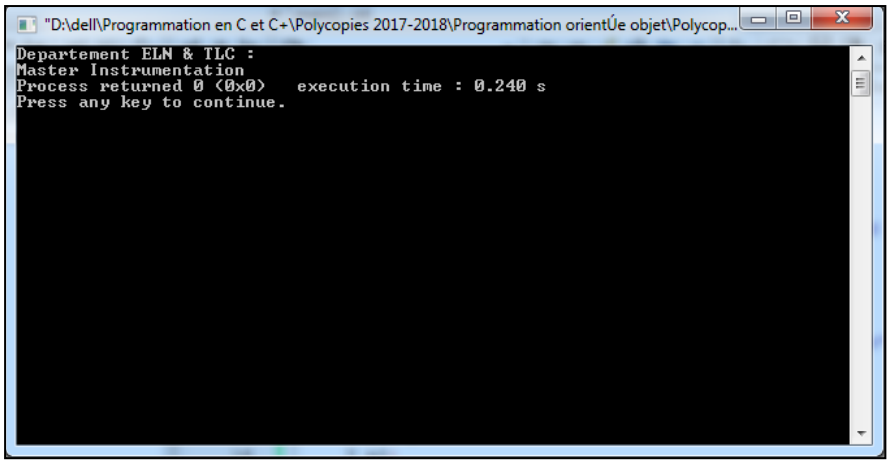
La redéfinition de fonctions (arrive au moment de l'exécution) est une fonctionnalité de la POO qui nous permet de redéfinir les méthodes d'une classe mère dans une classe dérivée avec la même signature.

**Exemple 4.2:**

```
#include<iostream>
using namespace std;
class X
{
public:
    virtual void afficher(){ cout << "Departement ELN & TLC :\n"; }
};
class Y: public X
{
public:
     void afficher(){ cout << "Master Instrumentation";};
};
int main()
{
   X p1,*ptr;
```

```
    Y p2;
    p1.afficher();
    ptr=&p2;
    ptr->afficher();
    return 0;
}
```

Résultat du programme est :




**6. Méthodes (fonctions membres) virtuelles**

Une fonction virtuelle est une fonction membre déclarée dans une classe de base et redéfinie par une classe dérivée. Lorsqu'on pointe vers un objet de classe dérivée à l'aide d'un pointeur ou d'une référence de la classe de base, on fait appelle à une fonction virtuelle de cet objet (de la classe de base) pour exécuter la fonction de la classe dérivée. Cela se fait en faisant précéder par le mot clé "**virtual**" la fonction membre de  la classe de base. Les fonctions virtuelles permettent de :

- Appeler la fonction membre correcte  d'un objet.
- Réaliser le polymorphisme dynamique ( l'appel de la fonction membre est effectuée au moment de l'exécution).

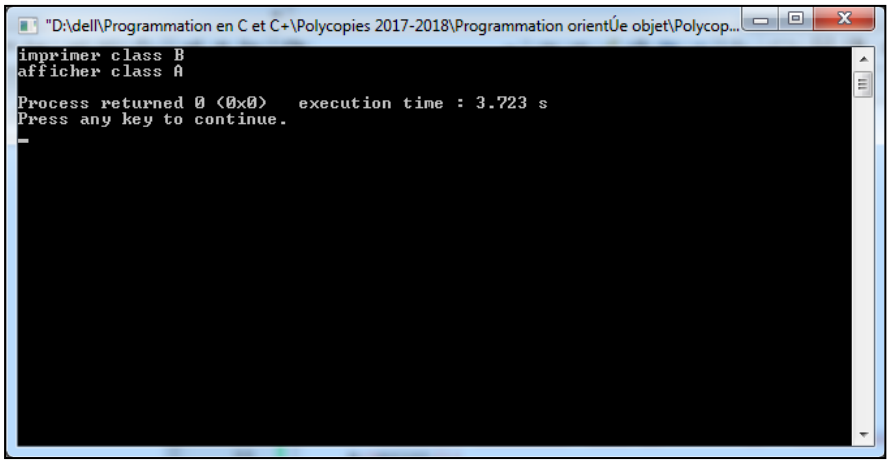
Le polymorphisme au moment de l'exécution est obtenu uniquement via un pointeur (ou une référence) de la classe de base. En outre, un pointeur de classe de base peut pointer sur les objets de la classe de base ainsi que sur les objets de la classe dérivée.

**Exemple 4.3:**

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

```
class A
{
public:
   virtual void print() {
   cout << "print class A" << endl;
  }
   void display() {
   cout << "display class A" << endl;
  }
};
class B : public A
{
public:
   void print () {
   cout << "print class B" << endl;
  }
   void display () {
   cout << "display class B" << endl;
  }
};
int main()
{
   A *a;
   B b;
   a = &b;
   // Fonction virtuelle, liée au moment de l'exécution
   a->print();
   // Fonction non virtuelle, liée à la compilation
   a->display();
}
```

Résultat de l'exécution du programme précédent :




Dans le code ci-dessus, le pointeur de classe de base « a » contient l'adresse de l'objet « b » de la classe dérivée.

**7. Méthode virtuelle pure**

Une fonction virtuelle pure est une fonction virtuelle pour laquelle nous avons que la déclaration de cette fonction. Une fonction virtuelle pure est déclarée en lui affectant 0 au moment de sa déclaration.

**Syntaxe:**

virtual type mon_fonct()=0;

**8. Classe abstraite**

Une classe abstraite est créée uniquement pour agir comme une classe de base (devant être héritée par d'autres classes). Elle contient au moins une fonction virtuelle pure. Les règles de conception d'une classe abstraite sont :

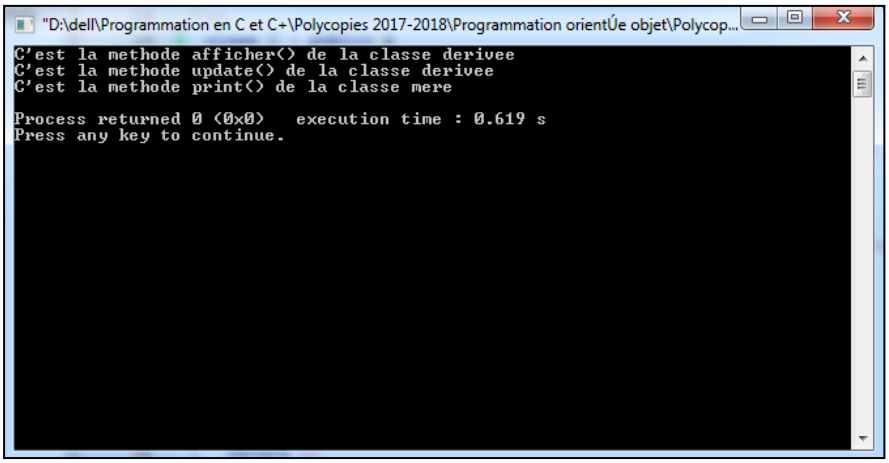
- Aucun objet d'une classe abstraite ne peut être créé.
- Elle est utilisée pour représenter des concepts généraux (par exemple, Forme, Animal), qui peuvent être utilisés comme classes de base pour des classes concrètes (par exemple, Cercle, Chien).
- Les pointeurs et les références vers une classe abstraite peuvent êtres déclarés.

**Exemple 4.4:**

```
#include <iostream>
using namespace std;
// Classe abstraite
class A
```

```
{
  public:
     virtual void afficher()=0; //Fonction virtuelle pure
     virtual void update()=0; //Fonction virtuelle pure
    void print()
    {
      cout << "C'est la methode print() de la classe mere" << endl;
    }
};
class B : public A
{
  public:
    void afficher()
    {
      cout << "C'est la methode afficher() de la classe derivee" << endl;
    }
    void update()
    {
      cout << "C'est la methode update() de la classe derivee" << endl;
    }
};
int main()
{
  B b;
  b.afficher();
  b.update();
  b.print();
  return 0;
}
```

Le résultat d'exécution du programme est :




## 9. Polymorphisme

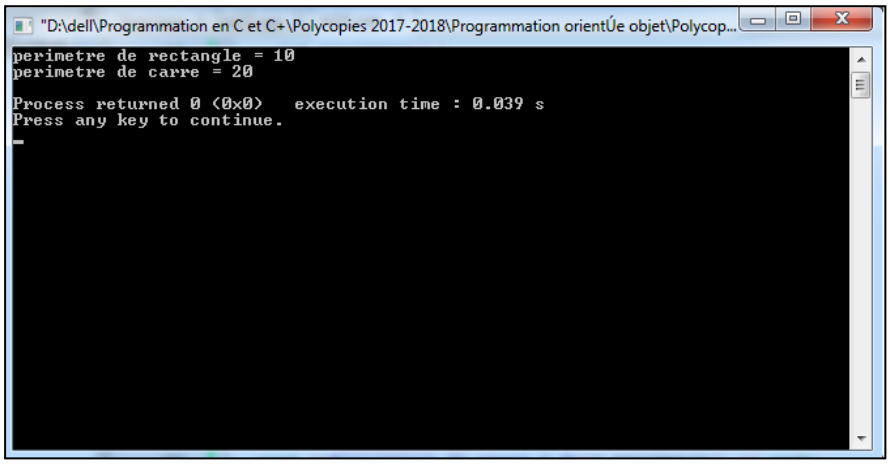
En général, le polymorphisme se produit lorsqu'il existe une hiérarchie de classes qui sont liées par héritage. Le polymorphisme est le troisième aspect essentiel d'un langage de programmation orienté objet, après l'abstraction des données et l'héritage. Il est implémenté en C++ avec les fonctions virtuelles (**virtual**) et l'héritage. Il permet au programmeur d'utiliser une méthode selon plusieurs manières, en fonction de son besoin.

**Exemple 4.5:**

```
#include <iostream>
using namespace std;
class forme {
  protected:
    int width, height;
  public:
    void update (int a, int b){
    width = a;
    height = b;
  }
};
class rectangle: public forme {
  public:
    int perimetre(){
    return (width + height)*2;
```

```
  }
};
class carre: public forme {
  public:
    int perimetre(){
    return (width+height)*4;
  }
};
int main () {
  rectangle r;
  carre c;
  forme * p1 = &r;
  forme * p2 = &c;
  p1->update(2,3);
  p2->update(2,3);
  cout <<"perimetre de rectangle = "<<r.perimetre() << endl;
  cout <<"perimetre de carre = "<<c.perimetre() << endl;
  return 0;
}
```

Résultat de l'exécution du programme précédent :




La fonction main() déclare deux pointeurs sur forme (nommés p1 et p2). Ces derniers se voient attribuer les adresses respectives de r et c, qui sont des objets de type

rectangle et carre. Ces affectations sont valides, car rectangle et carre sont des classes dérivées de forme.

# TP 1 : Maitrise du compilateur C++

## 1. Objectifs du TP

Familiarisation avec les notions de :

- Edition, compilation et édition des liens
- Les entrées sorties standards (clavier et écran)
- Déclarations des variables
- Erreurs syntaxiques
- Les structures de contrôle

## 2. Prise en main du logiciel de programmation

Dans cette section, on va décrire deux types de logiciels permettant la programmation en C++:

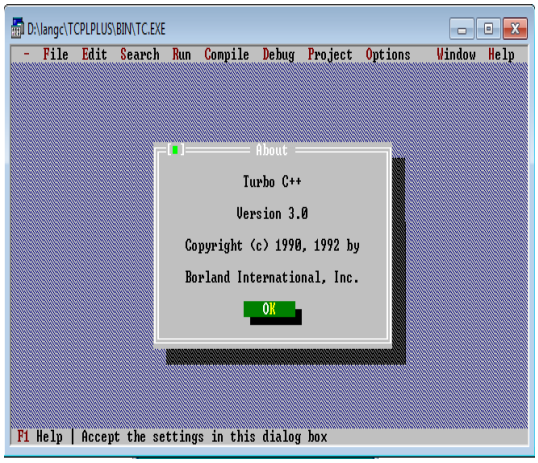
- Turbo C++ version 3 sous MS-DOS
- Code::Blocks 20.03 sous Windows

### 2.1 Description de Turbo C++ version 3

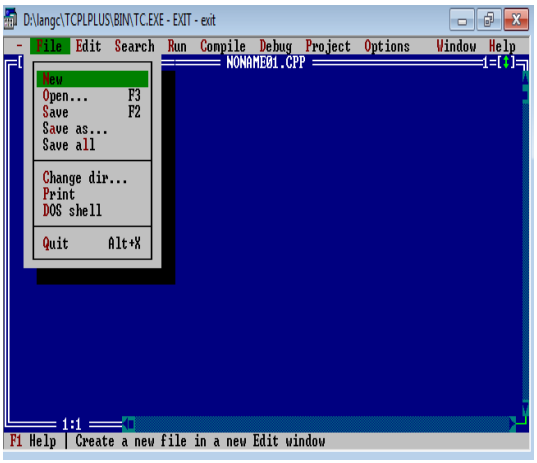
Le Turbo C++ version 3 est un environnement du développement intégré (IDE). Il est un logiciel de programmation en C++ sou MS-DS.

Un fois l'installation réussie de turbo C++.

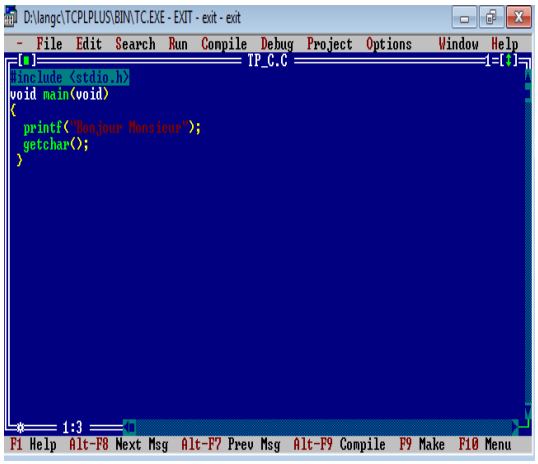
a. Lancer le **Turbo C++**, puis afficher le type de compilateur et la version du logiciel : **Alt+H→About**



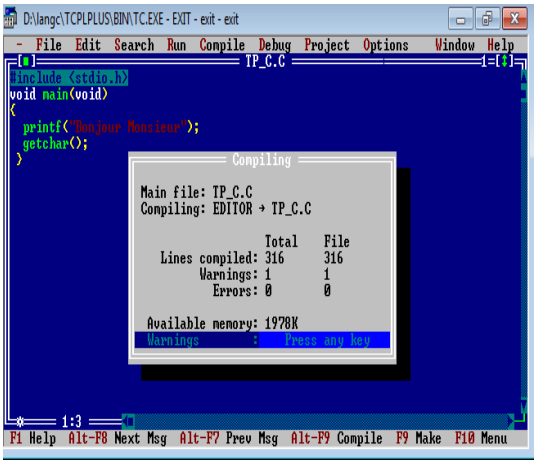

b. Créer un nouveau fichier, une fenêtre bleue va s'apparaître :**Alt+F→New**



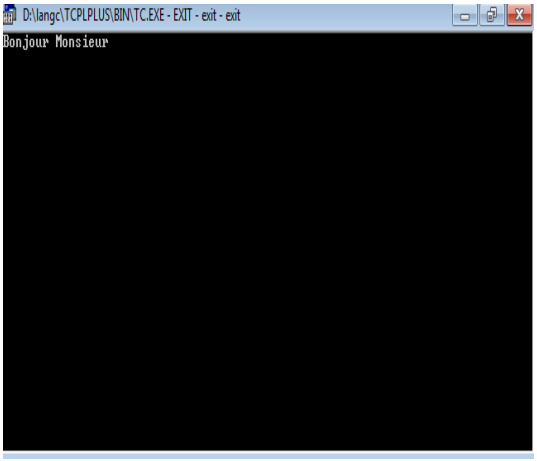

c. Pour commencer la programmation, écrire un programme dans l'éditeur de texte qui permet d'afficher le message "Bonjour Monsieur" sur écran, puis sauvegarder le fichier sous le nom "TP_1.cpp" (pour le sauvegarde : **Alt+F→Save**).




d. Compiler ce programme pour détecter s'il y a des erreurs, afin de les corriger : **Alt+C→Compile**.

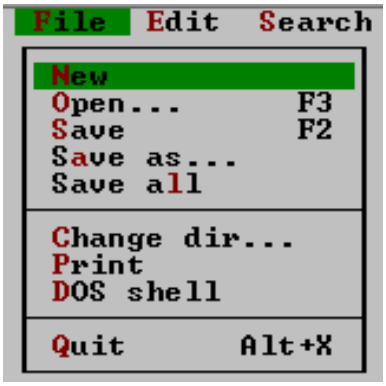


e. S'il n'y a pas d'erreur, exécuter le programme :**Alt+R→Run**.




### 2.1. Les différentes commandes de la fenêtre Menu

**File** (gérer les fichiers) : on créer un nouveau fichier (New), ouvrir un fichier déjà existant (Open, puis choisir le fichier dans la liste), sauvegarder le fichier actuel (Save ou Save as), changer le dossier courant (Change dir) ou bien quitter le Turbo C++ (Quit ou ALT X).



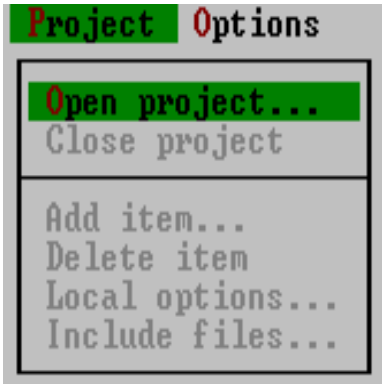

**Edit** : couper (Cut), copier (Copy), coller (Paste), remettre une ligne de texte dans son état initial (Undo), annuler les opérations effectuées par undo (Redo).

**Search** : recherche ou remplacement d'un texte dans tout le fichier (CTRL L pour rechercher le suivant).


**Run** : exécuter le programme. On peut exécuter le programme pas à pas (Trace into ou Step over). Ceci permet de voir dans quel ordre s'exécutent les instructions.


**Compile** : compiler pour corriger les erreurs et afin générer et exécuter le programme objet (nom_fich.exe).


**Project** : un projet contient la liste de fichiers nécessaires au fonctionnement d'un programme (fichiers séparés).




**Option** : on peut tout choisir : répertoires, niveau de messages d'erreur de compilation, mode 43 lignes (environnement preferences) et même les couleurs.


**Window** : déplacement, changement de taille de la fenêtre du programme.


### 2.2. Résumé sur les commandes utilisées

| Commandes | Description |
|---|---|
| **F1** | Afficher le help sur écran |
| **F2** | Sauvegarder le fichier courant |
| **F3** | Ouvrir un fichier |
| **F4** | Exécuter le programme à partir de la position courante du curseur |
| **F5** | Zoomer la fenêtre active |

| | |
|---|---|
| **F6** | Faire défiler les fenêtres qui sont ouvertes |
| **F7** | Exécuter le programme pas à pas |
| **F9** | Compiler le programme |
| **Alt+F4** | Quitter Turbo C++ vers DOS |
| **Ctrl+Ins** | Copier le texte sélectionné |
| **Shift+del** | Supprimer le texte sélectionné |
| **Shift+Ins** | Coller le texte copié |
| **Alt+F3** | Fermer la fenêtre active |
| **Alt+F5** | Afficher la fenêtre de DOS |
| **Ctrl+F5** | Change la position de la fenêtre active |
| **Ctrl+F9** | Exécuter le programme |
| **Alt+F9** | Compiler le programme |

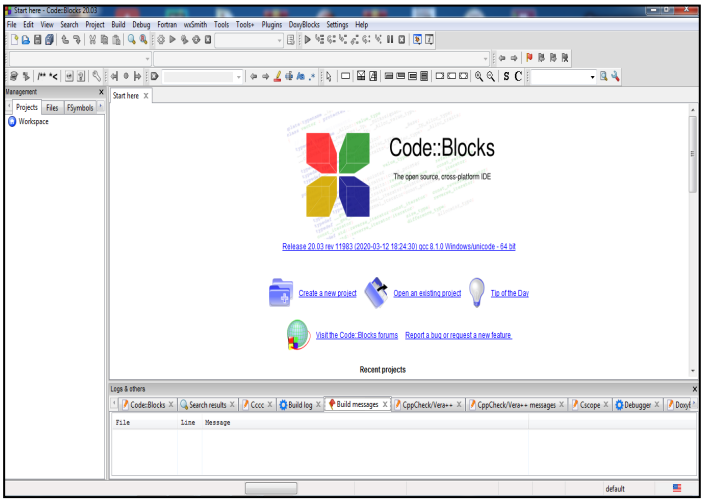
## 2.2. Description de Code::Blocks

Le logiciel code::blocks est un environnement du developpement intégré (IDE), ce qui signifie qu'il possède un éditeur de texte qui vous permettez d'écrire vos programmes sources en C++. Il est multiformes en particulier Windows et Linux. Il permet entre autre :

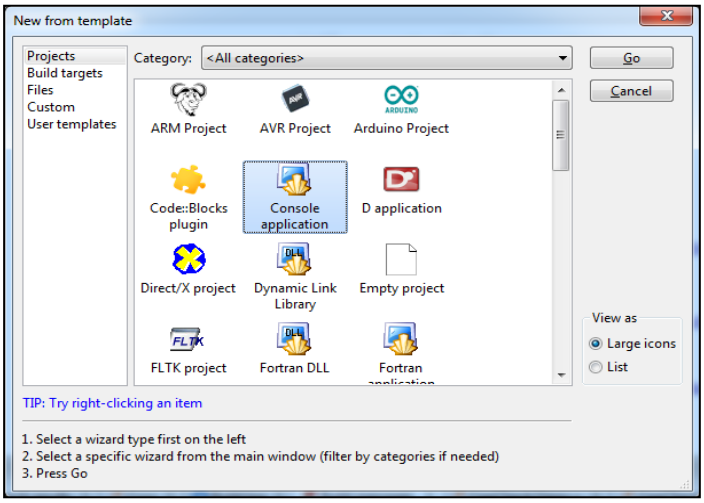
- La gestion des projets C++.
- La prise en compte de différents modèles de projet (console, graphique).
- L'appel intégré à différents compilateurs (par défault le compilateur GNU).

Une fois l'installation de code::blocks réussie.

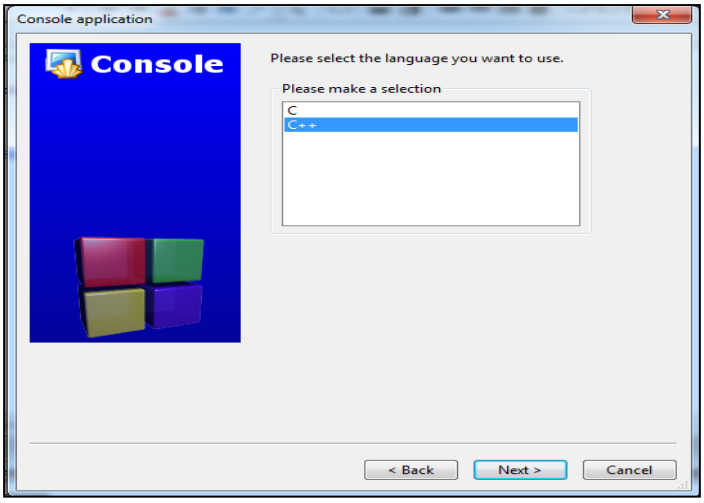
a. Lancer le code::blocks, une fenêtre s'apparaître :



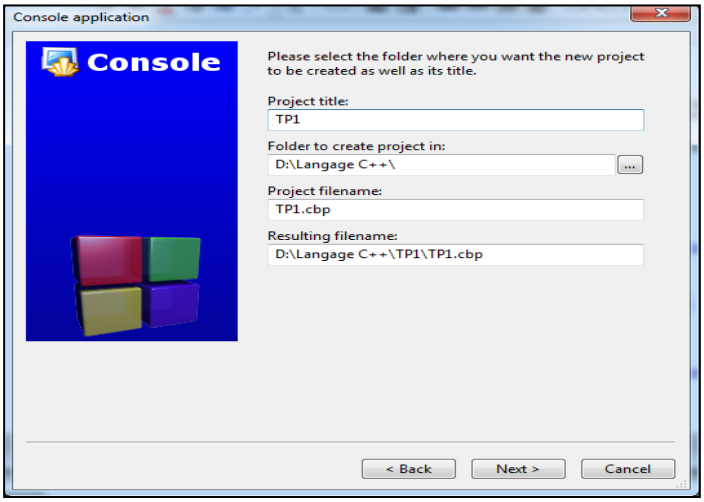
b. Créer un nouveau projet (cliquer sur **New**, puis **project** dans le menu **File**). Une fenêtre de dialogue s'apparaître et vous demande de choisir un modèle du projet (on choisi par exemple le modèle **console application**).



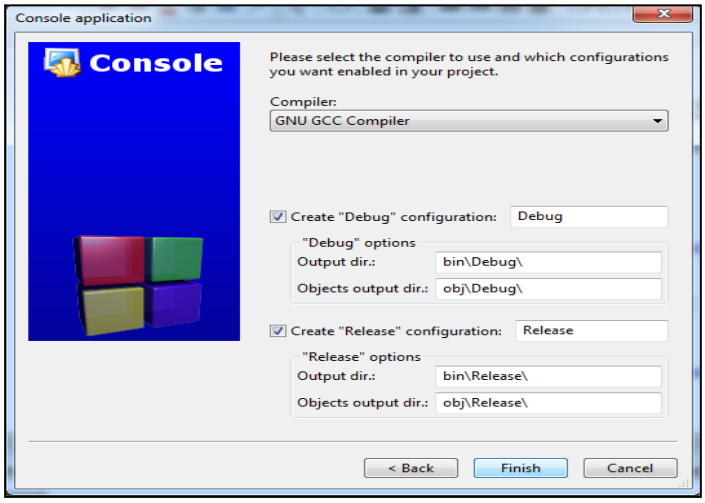

c. vous voulez développez une application C ou C++?



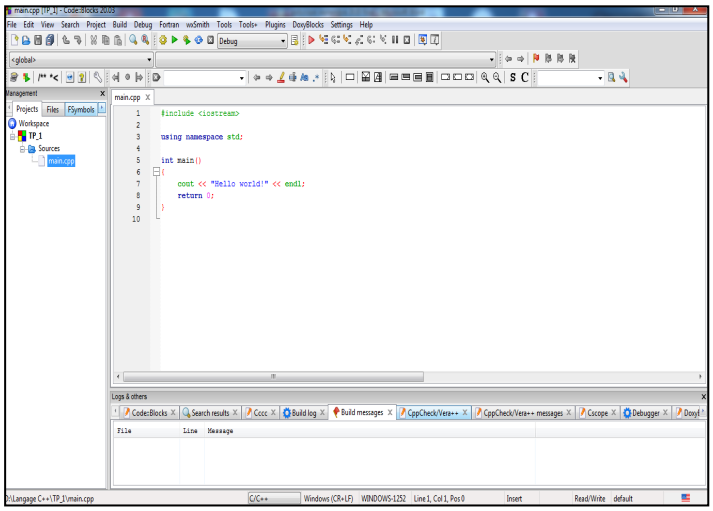

d. Donner le chemin d'accès et le nom du projet



e. Choisir le compilateur (par défaut GNU GCC).




f. Enfin, le projet est créer avec ses ressources.



## 3. Travail demandé

### 3.1. Manipulation 1

Soit le programme C ++ suivant :

```
#include <iostream>
using namespace std;
main()
{
int a,b,c,d ;
float g
cout<<"Réalisation de quelques opérations élémentaires \n";
cout<<"Donner les valeurs des deux entiers a et b :";
```

```
cin>>a>>b;
c=a*b;
d=a/b;
g=a%b;
cout<<"Les résultats obtenus sont : c="<<c<<"d="<<d<<"g="<g;
}
```

a. Après la saisie puis la compilation du programme, donner les erreurs syntaxiques et sémantiques.

b. Après la correction des erreurs, sauvegarder le programme sur le disque sous le nom " tp01.CPP", puis ré-effectuer la compilation. Exécuter le programme en donnant a=10 et b=3. Comparer entre les valeurs obtenues de "d" et de "g".

**3.2. Manipulation 2**

Traduire en programme C++, les deux algorithmes suivants :

*Algorithme1 :*

```
a,b :entrées (entiers) ;
s,r :sorties (entiers) ;
debut
ecrire('donner la valeur de a :') ;
lire(a) ;
ecrire('donner la valeur de b :') ;
lire(b) ;
s=0 ;
        Si (a>=b) Alors
                Faire
                r=a-b ;
                a=r ;
                  s=s+1 ;
                Fin Faire
        Sinon
                Faire
        Ecrire('le résultat de la division entière est = ',s) ;
```

Ecrire(‘le reste de la division entière est = ’,r) ;

Fin Faire

Fin.

*Algorithme 2 :*

Debut

Ecrire(‘donner un nombre entier’) ;

Lire(nb) ;

Si (nb>0) alors

Faire

Ecrire(‘le nombre’,nb,‘est un nombre entier positif’) ;

Fin Faire

Sinon si (nb=0) alors

Faire

Ecrire(‘le nombre’,nb,‘est =0’) ;

Fin Faire

Sinon Alors

Faire

Ecrire(‘le nombre’,nb,‘est nombre entier négatif’) ;

Fin Faire

Fin.

**3.3. Manipulation 3**

Un magasin offre une réduction sur achat pour les conditions suivantes :

- Si le montant d’achat est inférieur à 350 DA, il n’y a pas de réduction.
- Si le montant d’achat est compris entre 350 DA et 600 DA, le taux de la réduction est de 2%.
- Si le montant d’achat est supérieur à 600 DA, le taux de la réduction est 3%.

a. Proposer un organigramme qui permet de calculer le prix net à payer.

b. Traduire cet organigramme en programme C++.

**Sachant que :**

Net à payer = montant d’achat - montant de réduction

Montant de réduction=montant d’achat * taux de réduction

# TP 2 : Programmation en C++

## 1. Objectifs du TP

Maîtriser la manipulation des instructions de contrôles et itératives ainsi que les structures des données statiques et dynamiques.

## 2. Travail demandé

### 2.1. Manipulation 1

Ecrire un programme C++ qui calcule les racines carrées de nombres fournis en donnée avec les trois structures itératives (do…while, for et while). Il s'arrêtera lorsqu'on lui fournira la valeur 0. Il refusera les valeurs négatives. Rappelons que la fonction ***sqrt()*** fournit la racine carrée (***double***) de la valeur (***double***) qu'on lui fournit en argument.

### 2.2. Manipulation 2

a- Proposez un algorithme qui permet de calculer et d'afficher les résultats des expressions suivantes :

$\boldsymbol{S_1 = \sum_{i=1}^{N} i}$ $\quad S_2 = 2+4+6+\ldots+n$ **pour n>2** $\quad S_3 = 1+\frac{1}{2}+\ldots+\frac{1}{n}$ $\quad P = i!\,;\ i=\overline{0..N}$

b- Pour chaque expression, traduire l'algorithme correspondant en organigramme, puis en programme C++ pour les trois structures itératives (do…while, for et while) (pour N=10).

### 2.3. Manipulation 3

*Partie 1 :*

a- Proposer un algorithme qui permet de calculer la valeur d'une fonction ***F*** définie comme suit :

$$F(x) = \begin{cases} a.x^2 - c & \text{si } x < 0 \\ \frac{b}{x} & \text{si } x > 0 \\ 0 & \text{si } x = 0 \end{cases}$$

Avec *x* varie de ***d*** à ***f*** avec un pas ***h***.

b- Traduire cet algorithme en programme C++.

c- Même question pour $R = \prod_{i=1}^{k} (i^2 + \sum_{j=1}^{i} a_j{}^2)$

*Partie 2 :*

Proposez un organigramme qui permet de calculer et d'afficher les résultats des expressions suivantes :

$$S_1 = \sum_{i=1}^{N} \prod_{j=1}^{M} (i+j) \qquad , \qquad P_1 = \prod_{i=1}^{N} \sum_{j=1}^{i-1} (i*j)$$

Traduire cet organigramme en programme C++.

**2.4. Manipulation 4**

*Partie 1 :*

a- Proposez un organigramme permettant de remplir une liste de 20 nombres entiers, puis de trier/organiser cette liste par ordre croissant.

b- Traduire cet organigramme en programme C++. Puis modifier ce programme, pour classer ces nombres par ordre décroissant.

*Partie 2 :*

Ecrire un programme qui lit 10 nombres entiers dans un tableau avant d'en chercher le plus grand et le plus petit, en utilisant le formalisme tableau.

*Partie 3 :*

Soient deux tableaux t1 et t2 déclarés ainsi

**float t1[10], t2[ 10] ;**

Ecrire les instructions permettant de recopier, dans t1 tous les éléments positifs de t2, en complétant éventuellement t1 par des zéros.

**2.5. Manipulation 5**

*Partie 1 :*

Soit le tableau t déclaré ainsi :

**float t[3][4] ;**

Ecrire les instructions permettant de calculer dans une variable nommée som, la somme des éléments de t, en utilisant le formalisme des tableaux à deux indices.

*Partie 2 :*

Ecrire un programme qui permet d'élaborer une matrice ***A***(***N***, ***N***) dont les éléments ***a***(***i***,***j***) se calculent comme suit :

- Les éléments de la première colonne et de la diagonale principale doivent être égaux à un (‘1’). ***a(i,1) =1*** et ***a(i,i) =1***
- Les éléments abrités conjointement par la première colonne et la diagonale principale se déterminent selon la formule :

$$a(i,j) = a(i-1,j) + a(i-1,j-1)$$

- Le reste des autres éléments est identifié à zéro.

*Partie 3 :*

a. Ecrire un programme C ++ qui permet de calculer le produit matriciel de deux matrices A(N,M) et B(M,L) pour les deux cas suivant :

- les matrices A et B sont de type entier
- Les matrices A et B sont de type complexe

b. Ecrire un programme C qui permet de calculer le déterminant d’une matrice carrée E(4,4).

**2.6. Manipulation 6**

*Partie 1 :*

Soit une matrice A(N,M) dont l’élément est noté $a_{ij}$ , la matrice normalisée B(N,M) à ses éléments qui se calculent comme suit :

$$b_{ij} = \frac{a_{ij} - \bar{a}_j}{\sqrt{\sum_{k=1}^{N}\left(a_{kj} - \bar{a}_j\right)}} \qquad \text{Avec} \quad : \quad \bar{a}_j = \frac{1}{M}\sum_{k=1}^{N} a_{kj}$$

a. Proposer un algorithme qui permet de calculer les éléments de la matrice B(N,M).

b. Traduire cet algorithme en programme C++ pour (N=3et M=4).

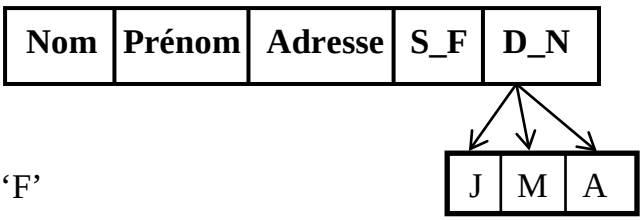
*Partie 2 :*

Soit une matrice A de n lignes et m colonnes. Ecrire un programme C++ qui permet de :

- Calculer la transposée de la matrice A
- D’afficher les éléments de la matrice $B=A^t$

*Partie 3 :*

On désire effectuer un recensement sur une population de 10 individus. Chaque individu est caractérisé par :




S_F=’M’ ou bien ‘F’

- Comment appelle-t-on le modèle de représentation qui permet de regrouper les données ainsi énumérées en une seule entité logique ?
- Ecrire un programme C++ permettant de remplir une liste de modèle décrit ci-dessus en mémoire.
- Modifier le programme précédent pour avoir une liste triée par ordre croissant selon le critère date de naissance (D_N).
- Puis proposer une modification du programme pour avoir une liste triée par ordre décroissant.

**Remarque :** au cours du changement (permutation), il faut tenir compte de l’ensemble des champs des modèles et non pas le champ « D_N » seulement.

# TP 3 : Classes et objets

## 1. Objectifs du TP

Maîtriser la manipulation des objets et les classes en C++.

## 2. Travail demandé

### 2.1. Manipulation 1

Ecrire un programme C++ qui permet de :

- Définir une classe appelée "point" ayant deux variables membres privées appelées "abscisse" et "ordonnee" de type réel.
- Créer deux objets de type "point", puis d'afficher les coordonnées de ces deux objets.

Faire cette manipulation de deux façons: avec et sans constructeur.

### 2.2. Manipulation 2

Ecrire un programme C++ qui permet de :

- Définir une classe "employe" avec les données membres privées suivantes: numéro employé, âge, années de service et le salaire.
- Créer deux objets de classe "employe", puis afficher leurs valeurs.
- Modifier le salaire d'un employé dont on donne le numéro, puis les afficher à nouveau.

Faire cette manipulation de deux manières: avec et sans constructeur.

### Manipulation 3

Taper sur votre machine le programme C++ suivant :

```
#include<iostream>
using namespace std;
class space
{
   int mcount;
public:
   space()
   {
    mcount=0;
```

```
    }
    space operator ++()
    {
       mcount++;
       return space(mcount);
    }
    void afficher()
    {
     cout<<"mcount = "<<mcount;
    }
};
void main()
{
   space objspace;
   objspace++;
   objspace.afficher();
}
```

a. Identifier et corriger les erreurs du programme.
b. Donner le résultat du programme.

**Manipulation 4**

Créer une classe polaire qui décrit un point dans le plan en utilisant, le rayon et l'angle des coordonnées polaires. Puis, utilisez l'opérateur + surchargé pour ajouter deux objets polaires. Notez que nous ne pouvons pas ajouter directement les valeurs polaires de deux points. Cela nécessite d'abord la conversion du point en coordonnées cartésiennes, puis l'ajout des coordonnées cartésiennes correspondantes et enfin la conversion du résultat en coordonnées polaires. Vous devez utiliser les formules trigonométriques suivantes:

$$\begin{cases} x = r * cos(\theta) \\ y = r * sin(\theta) \\ \theta = atan\left(\frac{y}{x}\right) \\ r = sqrt(x^2 + y^2) \end{cases}$$

# TP 4 : Héritage et polymorphisme

## 1. Objectifs du TP

Maîtriser la manipulation de l'héritage et le polymorphisme en C++.

## 2. Travail demander

### 2.1. Manipulation 1

Dans un établissement universitaire, on dispose des informations suivantes sur chaque employé et étudiant:

L'employé de l'administration ou autre possède :

- Numéro d'identification.
- Nom.
- Poste occupé par l'employé.
- Salaire.

Pour les enseignants, on dispose d'un champ supplémentaire :

- Matière enseignée

Pour les étudiants, on dispose des informations suivantes :

- Numéro d'identification.
- Nom.
- Titre du programme auquel l'étudiant est inscrit.

1. Ecrire un programme C++ qui permet de créer un objet de type employé, un objet de type enseignant et un objet de type étudiant, puis afficher les informations relatives aux objets ainsi crées.

**N.B:** Mettez tout ce qui est commun aux différentes classes dans la classes de base et le reste dans des sous classes (héritage).

### 2.2. Manipulation 2

Ecrivez un programme qui montre l'héritage multiple d'une classe "PointDeCouleur" avec 2 classes "Point" et "Couleur". La classe "Point" est définie sous forme d'un couple (x,y) qui sont les coordonnées d'un point.

La classe "Couleur" est définie sous la forme d'une chaîne de caractère. Vous n'oublierez pas de spécifier pour chacune des classes un constructeur et un destructeur.

### 2.3. Manipulation 3

Supposons qu'une banque gère deux types de comptes pour les clients, l'un appelé **compte d'épargne** et l'autre **compte courant**. le compte d'épargne offre des intérêts composés et des facilités de retrait mais pas de chéquier. Le compte courant fournit un chéquier et aucun intérêt. Les titulaires de comptes courants doivent également maintenir un solde minimum et si le solde tombe en dessous de ce niveau, des frais de service sont imposés.

Créez une classe **compte** qui stocke le nom personnalisé, le numéro de compte et le type de compte. De là dérivent les classes **cur_acct** et **sav_act** pour les rendre plus spécifiques à leurs besoins. Incluez les fonctions membres nécessaires pour accomplir les tâches suivantes:

- Acceptez le dépôt d'un client et mettez à jour le solde.
- Affichez la balance.
- Calculez et déposez les intérêts.
- Autoriser le retrait et mettre à jour le solde.
- Vérifiez le solde minimum, imposez une pénalité, nécessaire, et mettez à jour le solde.

**N.B:** N'utilisez aucun constructeur. Utilisez les fonctions membres pour initialiser les membres de la classe.

**Manipulation 4**

Modifier le programme de la manipulation précédente afin d'inclure des constructeurs pour les trois classes.

**Manipulation 5**

***Partie 1:***

Créez une classe de base appelée **forme**. Utilisez cette classe pour stocker deux valeurs de type **double** qui pourraient être utilisées pour calculer la surface des formes. Dérivez des classes spécifiques appelées **triangle** et **rectangle** à partir de la **forme** de base. Ajoutez à la classe de base, une fonction membre **get_data ()** pour initialiser les données membres de la classe de base et une autre fonction membre **display_area ()** pour calculer et afficher la surface des formes. Faites de **display_area ()** une fonction

virtuelle et redéfinissez cette fonction dans les classes dérivées en fonction de leurs besoins.

En utilisant ces trois classes, concevez un programme qui acceptera les dimensions d'un triangle ou d'un rectangle de manière interactive et affichera leur surface.

Rappelez-vous que les deux valeurs données en entrée seront traitées comme des longueurs de deux côtés dans le cas des rectangles, et comme base et hauteur dans le cas des triangles, et utilisées comme suit:

surface d'un rectangle=x*y

surface d'un triangle=1/2*x*y

***Partie 2:***

Etendez le programme de la première partie 1, afin d'afficher l'aire des cercles. Cela nécessite l'ajout d'une nouvelle classe dérivée "**cercle**" qui calcule l'aire d'un cercle. Rappelez-vous, pour un cercle, nous n'avons besoin que d'une seule valeur, son **rayon**, mais la fonction **get_data ()** dans la classe de base nécessite la transmission de deux valeurs (**Astuce:** faites en sorte que le deuxième argument de la fonction **get_data ()** soit par défaut avec une valeur nulle).

***Partie 3:***

1. Exécutez le programme ci-dessus avec les modifications suivantes:
   - Supprimez la définition de display_area () de l'une des classes dérivées.
   - En plus du changement ci-dessus, déclarez le display_area () comme virtuel dans la forme de classe de base.
2. Commentez les résultats obtenus dans chaque cas.

## Bibliographie


1. A. Boucena, "Introduction en langage C++", Edition O.P.U.
2. C. Delannay, "Exercices en langage C++", Edition Chihab-Eyrolles.
3. C. Delannay, "Programmer en langage C++", Edition Chihab-Eyrolles.
4. C. T. Wu and T. A. Norman, "An introduction to programming: An object-oriented approach with C++", the WCB/ McGraw-Hill Companies.
5. E. Balagurusamy, "Object oriented programming with C++", the McGraw-Hill Companies, Fourth edition.
6. J. Farrell, "Object-oriented programming using C++", Course Technology, Cengage Learning, Fourth edition.
7. M. Makhlouf, "Structures de données et algorithmes avec usage de langage C et C++", Edition Berti.
8. P. Aitken, "Apprenez le langage C++ en 21 jours", Edition S. SM.
9. P. Dax, "Langage C++". Edition Eyrolles.
10. R. Lafore, "Object-oriented programming in C++", Sams Publishing, Fourth edition.
11. S. Graïne, "Programmer en langage C++ avec exercices corrigés", Les Editions l'abeille.
12. T. Zhang, "Le grand proche : le langage C++", Edition S.SM.