MESRS

Année : 2019- 2020

Cours

Programmation Orientée Objet

Langage C++

MAMADOU Diarra
01/01/2020

Table des matières

1 Introduction à la Programmation Orientée Objets

1.1 Historique

1.1.1 Chronologie de la programmation

Il y a trois grandes époques :

L'âge du Chaos (**1950-1960**) **:** Langage machine et Assembleur

Temps de programmation considérable et programmes peu fiables

L'âge de la structure (1973-1980) : Langage évolué ou structuré

Temps de programmation considérable et problème de réutilisation

L'âge des objets (1990 et **+) :** Langage Objet (LO) ou Langage Orienté Objet (LOO)

Baisse du temps de programmation et réutilisation de logiciels

Exemples de LO ou LOO : VISUAL BASIC, Ada, C++, C#, D, Delphi (=Pascal orienté objet), Fortran 2003, Eiffel, Gambas 3, Java, Kylix, Objective-C, Objective Caml (ocaml), Perl, PHP (Depuis la version 4), Python, SmallTalk (totalement objet), Ruby, Julia.

1.1.2 Un peu d'histoire

La programmation orientée objets (P.O.O) ne date pas d'aujourd'hui, mais son succès reste récent. Elle remonte à la définition du langage de programmation Simula développé en Scandinavie dans les années 60.

Pascal et Simula 67, tous descendant de l'Algol-60 connaissent une évolution jusqu'en 1968. Ces langages étaient différents du point de vue démarche.

En Pascal, le programmeur se concentre sur les divers processus à réaliser, pour fabriquer ce système. Alors qu'en P.O.O le programmeur considère les entités qui existent dans le système et les modélise. Dans ce cas, le concepteur observe si les entités sont uniques ou existent en plusieurs exemplaires dans le système. Des collections de tels objets semblables sont des classes et il s'agira alors de déterminer le comportement caractérisant ces classes.

Une innovation fondamentale dans la P.O.O est l'introduction de l'héritage qui permet de définir une classe en l'étendant ou en spécifiant une classe existante. Et cela pour faciliter la tâche du programmeur car les classes créées par héritage constituent une hiérarchie de classes permettant ainsi une structure significative sur laquelle on peut construire et également comprendre un système logiciel.

Chaque classe constitue une abstraction pour servir à la construction de d'autres abstractions.

En définitive, dans la P.O.O, la spécification de données (objet) précède la définition des procédures (comportement ou méthodes).

En 1970, la P.O.O dans le souci de convivialité intègre la prise en compte des interfaces homme - machine (personne- système)

L'on peut citer smalltalk développé par le centre de recherche Park de Xerox empruntant des formalismes orientés objets dans la conception des SE, OS/2.

En 1980 est créé Object Pascal par Apple computer. Malheureusement cette version ne connaîtra pas trop de succès car il manque d'uniformité rendant ainsi laborieuse la tache du programmeur.

Par expansion, la P.O.O a connu de nouveaux langages comme C++, Eiffel, Modula-3, Java, Ada 95.

Dans le cadre de notre cours, nos exemples seront faits en C++ développé au laboratoire Bell (d'ou est sorti Unix et C) par Bjarne Stroustrup.

1.2 Définition et caractéristiques de la P.O.O

1.2.1 Quelques définitions

La P.O.O est un ensemble de nouveaux langages de programmation, une nouvelle façon de penser, un paradigme de programmation car il permet de comprendre un ensemble de théories, de normes et de méthodes qui, ensemble représente une façon d'organiser la connaissance ou tout simplement une façon de voir le monde.

Comme autre paradigme de programmation, l'on peut citer :

- Programmation impérative ou procédurale (Pascal)
- Programmation logique (Prolog)
- Programmation fonctionnelle (Lisp, Ml)
- Programmation graphique événementielle (VB)

Les logiciels modernes sont développés à l'aide du C++, Java en général à cause de leur portabilité.

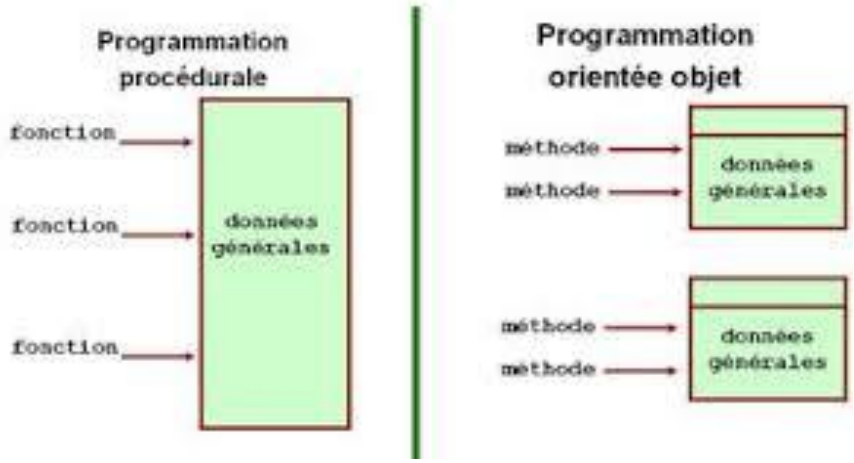
A côté de ces programmes, il existe la programmation graphique événementielle telle que Visual Basic, Visual C++, Delphi.

Cette dernière utilise des objets (briques logiciels), le graphique (fenêtres, icônes, menus, ...) et les événements (sollicitation de souris, clavier …)

De façon commune, dans l'exécution d'un programme, le code source est exécuté ligne par ligne par un interpréteur et ensuite traduit en langage machine par un compilateur.

Donnons quelques définitions dans le contexte orienté objets.

D'abord la P.O.O est une méthode d'implantation dans laquelle les programmes sont organisés comme une collection d'objets, chacun étant une instance d'une classe, lesquelles font partie d'une hiérarchie de classes unifiées via des relations d'héritage.



Programmation - A. Cohen - Programmation orientée objet

Classes : une classe est une abstraction représentant une collection d'objets ayant des caractéristiques semblables. En P.O.O, les données et les fonctions sont fusionnées au sein d'un objet informatique appelé classe.

Objets : un objet est une instance spécifique de sa classe, qui lui associe un ensemble d'opérations, il possède un état, on le repère par un nom et peut être observé à partir de sa spécification ou à partir de sa réalisation.

En somme, un objet est une entité qui peut :

- Changer d'état
- Se comporter de façon discernable
- Être manipulé par diverses formes de stimuli
- Être en relation avec d'autres objets

Cela se résume par le fait qu'un objet existe, occupe de l'espace, possède un état, possède des attributs et exhibent des comportements.

En définitive, un objet est défini via sa classe qui détermine tout à propos de l'objet. Par exemple on peut créer un objet **kone** de la classe **etudiant.**

La classe **etudiant** va donc définir qu'est-ce qu'un objet de type **etudiant** ?

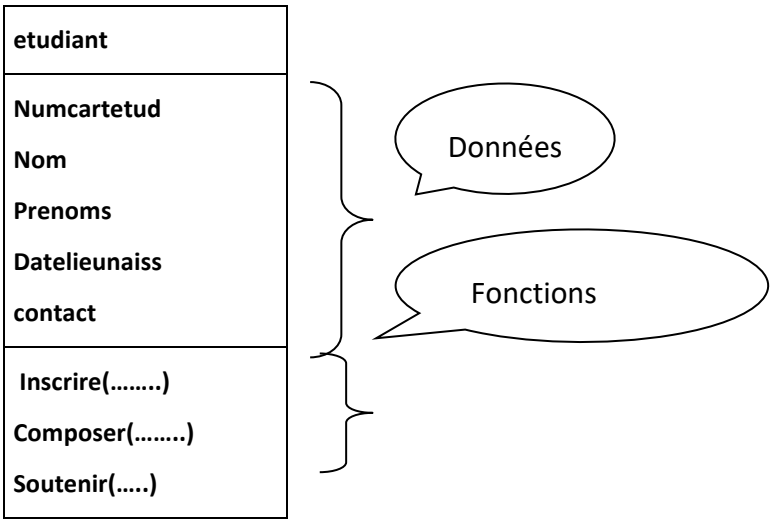
Et tous les messages sous lesquels, un objet de type **etudiant** peut interagir.

Chaque langage a sa méthode qui est simplement l'action à effectuer suite à la réception d'un message. Souvent, les paramètres sont fournis avec les messages. L'on pourrait utiliser le message '' **inscrire** '' ; ce message devrait contenir des paramètres **date_inscription, classe, frais ?** Supposons une inscription d'étudiant que l'on peut représenter par le formalisme suivant :

kone.inscrire('12/03/2007', 'licence 3 miage', 200 000);

cad : objet.fonctionmembre(listes paramètres) ;

Exemple de classe :



1.3 Analyse orientée objet

Le but de l'analyse orientée objet, comme pour toute autre analyse, est d'obtenir une compréhension de l'application : une compréhension qui dépend uniquement des exigences fonctionnelles du système.

L'analyse orientée objet contient, dans un certain ordre, les activités suivantes :

- Trouver l'objet.
- Organiser les objets.
- Décrire comment les objets interagissent.
- Définir les opérations des objets.
- Définir les objets en interne.

1.4 Caractéristiques de la POO

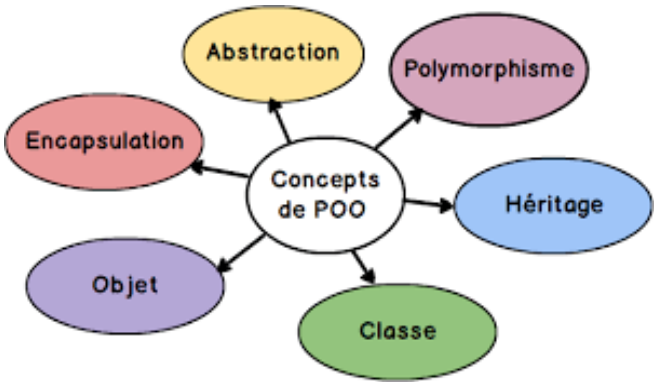
1.4.1 Solution

3ème amélioration : l'approche objet

Une entité autonome qui regroupe un ensemble de propriétés (données) cohérentes et de traitements associés.

Une telle entité est un objet et constitue le concept fondateur de l'approche objet.

En résumé, centraliser les données d'un type et les traitements associés, dans une même unité physique, permet de limiter les points de maintenance dans le code et facilite l'accès à l'information en cas d'évolution du logiciel.

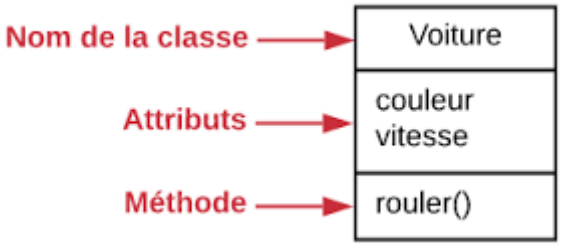


Concepts de la POO

1.4.2 Concept de classe

C'est le constructeur de base en POO, c'est la mise en œuvre d'un Type Abstrait (TA).

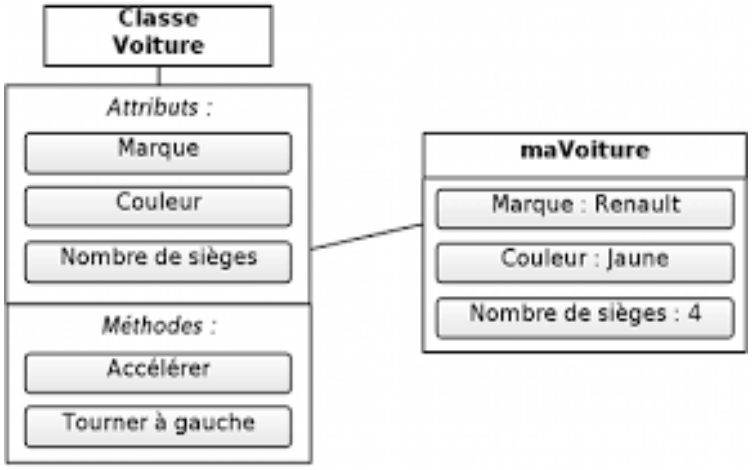
Une classe est un type de données abstrait, caractérisé par des propriétés (attributs et méthodes) communes à des objets et permettant de créer des objets possédant ces propriétés.



Classe voiture

1.4.3 Concept d'objet

Un objet ou instance de classe est la mise en œuvre du TA ou d'une classe. Tout comme une variable est un représentant d'un type, un objet est aussi un représentant d'un TA ou d'une classe.



Objet instanciation de classe

1.4.4 Encapsulation (ou masquage de l'information)

Lorsque l'objet est parfaitement bien écrit, il introduit la notion fondamentale d'Encapsulation des données. Ceci signifie qu'il n'est plus possible pour l'utilisateur de l'objet, d'accéder directement aux données : il doit passer par des méthodes spécifiques écrites par le concepteur de l'objet, et qui servent d'*interface* entre l'objet et ses utilisateurs. L'intérêt de cette technique est évident. L'utilisateur ne peut pas intervenir directement sur l'objet, ce qui diminue les risques d'erreur, ce dernier devenant une "*boîte noire*".

Elle permet de :

- protéger les données d'un objet contre une manipulation dangereuse par un autre objet ;
- rendre plus aisé la manipulation des données via une interface bien définie (partie publique) ;
- dissimuler les détails de son implémentation (partie privée) : Analogie avec un circuit intégré.

1.4.5 Instanciation

Opération de création dynamique d'un exemplaire d'une classe (objet).

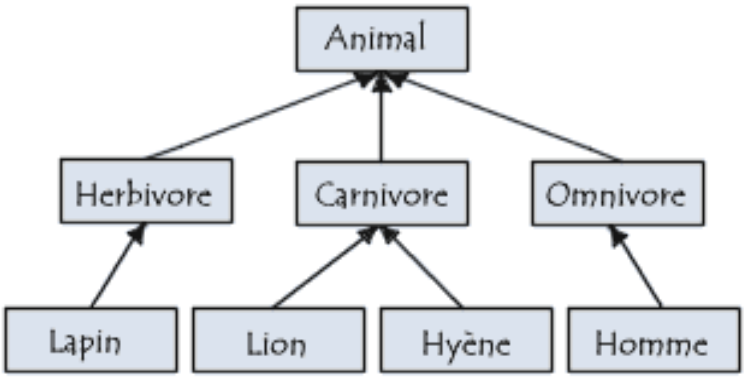
Communication par envoi de messages

Envoyer un message à un objet, c'est lui dire ce qu'il doit faire. La réponse d'un objet à la réception d'un message est une activation de méthode.

1.4.6 Héritage

L'héritage permet de décrire une nouvelle classe (par extension, spécialisation) à partir d'une ou plusieurs autres classes.

Son intérêt est la réutilisation de classes existantes.

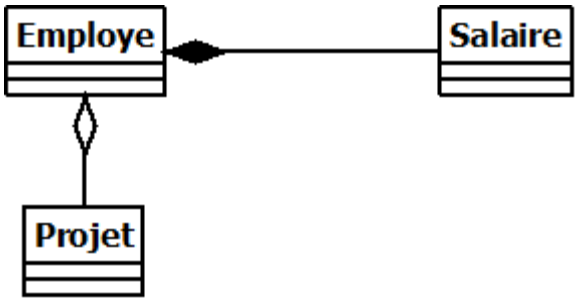


Héritage

1.4.7 Agrégation

Relation entre deux classes, spécifiant que les objets d'une classe sont des composants de l'autre classe.

Exemple : Classe Voiture composée des classes Roues, Châssis et Carrosserie



Agrégation et composition

1.4.8 Généricité

La généricité est la paramétrisation de la définition d'une fonction ou d'une classe.

Intérêt : permettre l'écriture de fonctions ou de classes flexibles et réutilisables.

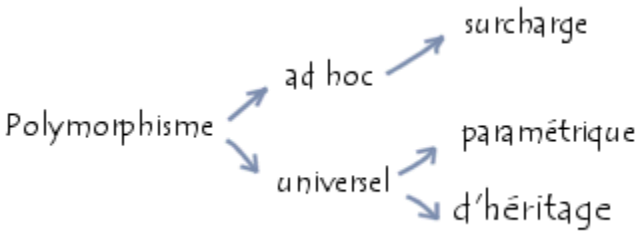
1.4.9 Polymorphisme

Mot grec signifiant "qui se présente sous différentes formes". C'est la capacité donnée à une même méthode de s'exécuter différemment suivant le contexte de sa classe.

Il permet la définition d'entités flexibles pouvant référencer des objets de formes variables à l'exécution.

On distingue trois types de polymorphisme :

- Le polymorphisme ad hoc (également *surcharge* ou en anglais *overloading*)
- Le polymorphisme paramétrique (également *généricité* ou en anglais *template*)
- Le polymorphisme d'héritage (également *redéfinition*, *spécialisation* ou en anglais *overriding*)



Formes de polymorphisme

Exemple de méthode objet

- La méthode HOOD (Hierachical Object Oriented Design)
- La méthode OMT (Object Modeling Technic)
- Le Processus Unifié et UML (Unified Modeling Language)

1.5 Langages de POO

1.5.1 Présentation du C++

Il existe plusieurs langages de P.O.O majeures utilisés de nos jours. Mais les langages orientés objets commerciaux dominants sont peu nombreux. Ce sont C++**, Smalltalk** et **Java**

C++ : c'est une version orientée objets du langage C. Il est compatible avec le C (il englobe entièrement le C). Ce qui permet d'incorporer du code C existant dans les programmes C++.

Les programmes C++ sont rapides et efficaces, c'est ce qui lui a valu les meilleures qualités pour devenir un langage extrêmement populaire.

Le **C++** utilise le **Compile-time binding** ce qui signifie que le programmeur doit spécifier la classe spécifique d'un objet ou au moins, la classe la plus générale à laquelle appartient l'objet au moment de la compilation. Cela permet un temps d'exécution efficace et une taille de code plus petite, il y a un compromis dans la puissance de l'utilisation des classes.

Le **C++** est devenu si populaire que la majorité des nouveaux compilateurs C sont en réalité des compilateurs **C/ C++.**

1.5.2 Java

Il est le plus récent et le plus connu des langages de P.O.O. Il a pris d'assaut le monde du logiciel par le lien très étroit avec Internet et les navigateurs web. Il est conçu comme langage portable qui peut tourner sur tout ordinateur branché sur Internet via un navigateur.

Ainsi, il est très prometteur pour devenir le standard de la programmation d'application Internet et intranet.

Java est un mélange de C++ et Smalltalk. Il a l'avantage de ne pas avoir de pointeurs et possède une **garbage collector** qui est une caractéristique qui libère le programmeur d'allouer et désaouller explicitement la mémoire.

Il s'exécute sur une machine virtuelle. Il s'agit d'un logiciel intégré à votre navigateur qui exécute les mêmes bytes codes Java peu importe le type d'ordinateur que vous possédez.

En résumé, la P.O.O offre un puissant modèle pour l'écriture de logiciels d'ordinateurs.

Cette méthode accélère le développement de nouveaux programmes et améliore la maintenance et la réutilisation des logiciels.

Le C++ offre une transition plus facile à partir du C mais demande la maîtrise de la conception orientée objets pour en faire bon usage de cette nouvelle technologie.

2 Classes et fonctions membres

2.1 Notion de classe

2.1.1 Utilisation de l'instruction struct

Une classe est une généralisation de la notion de type définie par l'utilisateur dans lequel se trouve associé des données (**données membres**) et des méthodes (**fonctions membres**).
En pratique le type **struct** sera rarement employé sous la forme généralisée (il s'agit d'une classe où les données ne sont pas encapsulées).
Pour la déclaration d'une structure, l'on peut écrire :

```
/* déclaration d'une structure */.
struct point
 {
/* déclaration classique des données */
 int x ;
int y ;
/* déclaration des fonctions membres */
void initialise (int, int) ;
void deplace (int, int) ;
void affiche() ;
} ;
```

Ici on définit une structure point telle que :
- x et y sont des champs où des membres de la structure **point**.

Initialise, déplace et affiche sont des fonctions membres ou méthodes de cette structure de façon logique après la déclaration de la structure, on définit les différentes fonctions membres en utilisant le symbole **::** appelé **opérateur de résolution de portée** ou simplement **opérateur de portée.**
D'ou l'on écrit :

```
void point :: initialise(int abs, int ord)
        {
                  x = abs ;
                y = ord ;
        }
void point :: deplace(int dx, int dy)
        {
                  x+ =dx ;
                y += dy ;
        }
void point :: affiche( )
        {
                  cout<<'' je suis en'' <<x <<'' ''<<y<<''\n''<<endl ;
```

}

Remarque

Les opérateurs de portée permettent de spécifier que les fonctions ci-dessus appartiennent à la structure point.

- **cout** permet d'afficher une information mais l'on peut forcer l'affichage si le besoin se fait sentir ;
- **cin** permet d'entrer une information ;
- **\n** peut s'écrire **endl**, permet de revenir à la ligne ;

Ces différentes fonctions membres peuvent être appelées dans un programme principal appelé **main**, par exemple a.initialise (-3,2) ; signifie l'envoi d'un message initialise accompagné des informations –3 et 2 à l'objet a.

Ainsi l'accès aux membres se fait par l'opérateur '**.**'

Pour manipuler les différentes fonctions membres, l'on peut écrire le programme suivant :

```
 main()
 {
point a, b ;
a.initialise(5,2) ; a.affiche() ;
b.initialise(-1,1) ; b.affiche() ;
a.deplace(-2,4) ; a.affiche() ;
getch() ;
return 0 ;
}
```

à l'exécution

```
je suis en  5   2
je suis en  3   6
je suis en  1   -1
```

2.2 Classes

struct est un cas particulier de la classe car dans cette structure toutes les fonctions membres sont publiques.

La déclaration d'une classe est voisine d'une structure, il suffira de préciser les membres publics et privés et remplacer simplement **struct** par **class.**

Il est indispensable de préciser la bibliothèque de gestion des entrées sorties qui est ''**iostream.h** ''.

Comme exemple, nous utiliserons l'exemple de la classe point.

```
#include<iostream>
#include<conio.h>
using namespace std;
/*declaration d'une classe */
```

```
class point
 {
    /*declaration des members privés */
   private:/*facultatif*/
int x ;
int y ;
    /*declaration des membres  publics*/
public :
void initialise (int, int) ;
void deplace (int, int) ;
void affiche () ;
} ;
void point :: initialise (int abs, int ord)
 {
   x = abs ;
           y = ord ;
  }
void point ::deplace(int dx, int dy)
         {
x+ =dx ;
                  y += dy ;
         }
void point ::affiche( )
         {
cout<<'' je suis en'' <<x <<'' ''<<y<<endl;
         }
/*programme principal */
  main()
 {
     //clrscr() ;//system(''cls'') ;
point a, b ;
a.initialise(5,2) ;a.affiche() ;
b.initialise(-1,1) ;b.affiche() ;
a.deplace(-2,4) ;a.affiche() ;
getch() ;
return 0 ;
```

```
}
```

Commentaire

En P.O.O on dit que a et b sont des instances de la classe point ou encore des objets du type point.

S'il n'y a pas de private ou public dans une classe, ses membres sont considérés privés donc inaccessibles et cela est inutile.

Si l'on rend tous les membres d'une classe publique, on obtient l'équivalence d'une structure.

Ainsi les deux codes suivants sont équivalents.

Il existe un troisième mot réservé qui est protected désignant une situation intermédiaire entre private et public (cela est généralement utilisé dans les classes dérivées)

Résumé

Une classe est une structure dont les attributs sont des données ou des méthodes.

Un objet ou une instance est un exemplaire de cette structure. De façon pratique on déclare la classe, on définit les fonctions membres et on déclare les objets dans un programme principal dans lequel ils seront manipulés via les fonctions membres.

Autre exemple de classe

```
#include <iostream>
#include <string.h>
#include <conio.h>
using namespace std;
class etudiant
{
char nom[10];
int age;
char classe[10];
public :
void saisir (char[ ], int, char[ ]);
void afficher( );
};
void etudiant::saisir(char sn[10], int a, char sc[10])
{
```

```
strcpy(nom, sn);
age =a;
strcpy(classe, sc);
}
void etudiant::afficher( )
{
cout<<"Name=\t"<<nom<<endl;
cout<<"Age=\t"<<age<<endl;
cout<<"Class=\t"<<classe<<endl;
}
main( )
{
etudiant s1, s2;
char st[10];
int sa;
char scls[10];
//clrscr( );
cout<<"Saisir les infos de l etudiant 1"<<endl;
cout<<"Donner le nom:"<<endl;
cin>>st;
cout<<"Donner l age:"<<endl;
cin>>sa;
cout<<"Donner la classe:"<<endl;
cin>>scls;
s1.saisir(st,sa,scls);
cout<<"Saisir les infos de l etudiant 2"<<endl;
cout<<"Donner le nom:"<<endl;
cin>>st;
cout<<"Donner l age:"<<endl;
cin>>sa;
cout<<"Donner la classe:"<<endl;
cin>>scls;
s2.saisir(st,sa,scls);
```

```
cout<<"*** Affichage des informations des tudiants ****"<<endl;
cout<<"\n\t etudiant 1"<<endl;
s1.afficher( );
cout<<"\n\t etudiant 2"<<endl;
s2.afficher( );
getch( );
return 0;
}
```

2.3 Fonctions membres

Nous nous contenterons de définir quelques propriétés des fonctions membres.

2.3.1 Surdéfinition des fonctions membres

Il y a surdéfinition lorsqu'un identificateur désigne plusieurs membres.

Pour cela ces fonctions doivent différer par la liste des types de leurs arguments.

Exemple : surdéfinition d'une fonction affiche()

```
#include<iostream>
#include<conio.h>
using namespace std;
void affiche( int i)
        {
         cout<<"entrer valeur"<<i<<endl;
      }
void affiche( char c )
 {
   cout<< '' caractere ''<<c<<endl ;
 }
void affiche( double d )
 {
   cout<< '' reel ''<<d<<endl ;
 }
void affiche( string txt, int n )
 {
  for(int j=0 ;j<n ;j++)
   cout <<txt[j];
   cout<<endl ;
 }
main()
 {
```

```
    //clrscr() ;//sytem(''cls'') ;
    affiche(3)      ;
    affiche('x' );
    affiche(1.0);
    affiche(''bonjour tout le monde'',7);
  getch() ;
  return 0 ;
  }
```

Remarque :

Il est possible de diminuer le nombre de fonctions surdéfinies. Dans ce cas on utilise le **passage d'arguments par défaut.**

2.2.2. Arguments par défaut

Dans l'exemple point, on peut modifier la classe point en utilisant qu'une seule fonction affiche(). Dans cet exemple, elle a un seul argument de type chaîne et sa valeur par défaut est la chaine vide.

```
#include<iostream>
#include<conio.h>
using namespace std;
class point
{
int x, y ;
public: // surdefinition de la fonction point
point();
point(int);
point(int, int);
void affiche (char*);
};
point :: point()
{
x=0; y=0;
}
point :: point( int abs)
{
x=y=abs;
}
point :: point( int abs, int ord)
{
x = abs; y =ord;
```

```
}
void point:: affiche(char* message)
{
cout<<message<<"je suis en  "<< x <<" "<< y <<"\n";
}
main()
{
point a;
a.affiche(‘’  ‘’);
point b(5);
b.affiche("pointb");
point c(3,12);
c.affiche("Bonjour Monsieur");
getch();
return 0;
}
```

Exercice

Ecrire un programme qui demande à l'utilisateur de donner un entier, une chaîne de caractère et réel.

Dès que l'utilisateur rentre ces trois informations, la machine affiche cela autant de fois suivant la valeur d'entier entrée par l'utilisateur.

2.3.2 Fonction membre en ligne

On peut utiliser les fonctions en ligne pour accroître l'efficacité d'un programme dans les cas de fonctions courtes. Il y a deux manières de les déclarer.

1ère méthode

On définit la fonction dans la déclaration de la classe dans ce cas et il est inutile d'utiliser le mot inline.

Exemple :

```
#include<iostream.h>
#include<conio.h>
        class point
                {
        int x,y;
        public:// fonction point sont en ligne
        point() {x=0;y=0};
       point(int abs){x=y=abs ;}
```

```
        point(int abs, int ord){x=abs ;y=ord ;}
        void affiche(char*=" ");
       };
```

Exemple :

```
#include<iostream>
#include<conio.h>
using namespace std;
/*declaration d'une classe */
class point
 {
   /*declaration des members prives */
  private:/*facultatif*/
int x ;
int y ;
   /*declaration des membres  publics*/
           public :
void initialise (int abs, int ord) // fonction en ligne
{
  x =abs ;
          y = ord ;
 }
void deplace (int dx, int dy) // fonction en ligne
 {
x+=dx ;
y += dy ;
         }
void affiche () // fonction en ligne
 {
cout<<" je suis en " <<x <<" "<<y<<endl;
         }
} ;
/*programme principal */
  main()
 {
point a, b ;
a.initialise(5,2) ;a.affiche() ;
b.initialise(-1,1) ;b.affiche() ;
a.deplace(-2,4) ;a.affiche() ;
```

```
getch() ;
return 0 ;
}
```

2ème méthode

On procède comme pour les fonctions ordinaires mais on utilise le mot clé **inline** dans la déclaration des fonctions et dans et dans l'en-tête.

Exemple :

```
#include<iostream.h>
#include<conio.h>
        class point
                {
        int x, y;
        public:
        inline point();
        inline  point(int);
       inline   point(int,int);
        void affiche(char*=" ");
          };
inline point ::point(){……}
```

Exemple :

```
#include<iostream>
#include<conio.h>
using namespace std;
/*declaration d'une classe */
class point
 {
   /*declaration des members prives */
  private:/*facultatif*/
int x ;
int y ;
   /*declaration des membres  publics*/
          public :
inline void initialise (int, int) ;
inline void deplace (int, int) ;
inline void affiche () ;
} ;
inline void point :: initialise (int abs, int ord)
 {
```

```
   x = abs ;
          y = ord ;
  }
inline void point ::deplace(int dx, int dy)
        {
x+=dx ;
                y += dy ;
        }
inline void point ::affiche( )
        {
cout<<" je suis en " <<x <<"  "<<y<<endl;
        }
/*programme principal */
  main()
 {
  point a, b ;
a.initialise(5,2) ;a.affiche() ;
b.initialise(-1,1) ;b.affiche() ;
a.deplace(-2,4) ;a.affiche() ;
getch() ;
return 0 ;
}
```

2.3.3 Objet transmis en argument d'une fonction

Dans les manipulations des objets, l'on peut les transmettre en argument de fonction c'est à dire tout simplement que les fonctions acceptent les types de données complexes.

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


            class point
                    {
           int x, y;
           public:
           point(int abs=0, int ord=0)
            {
                    x=abs; y=ord;
              }
           int coincide(point);
          };
             int point::coincide(point pt)
```

```
            {
            if ((pt.x==x)&&(pt.y==y))return 1;
            else return 0;
           }
    main()
           {
        point a, b(1), c(1,0);
        cout<<"a et b:" <<a.coincide(b)<<"ou"<<b.coincide(a)<<"\n";
        cout<<"b et c:" <<b.coincide(c)<<"ou"<<c.coincide(b)<<"\n";

    getch() ;
   return 0 ;
       }
```

à l'exécution on a :

a et b: 0 ou 0

b et c : 1 ou 1

2.4 Transmissions des objets en argument

2.4.1 Transmissions de l'adresse d'un objet

On peut transmettre une valeur qui se trouve à une adresse et qu'il faut interpréter dans la fonction. Dans ce cas on utilise l'opérateur d'indirection ***.**

Au niveau du **main()** l'on serait obligé d'utiliser l'opérateur **&.**

L'on peut modifier l'exemple précédent par ceci :

```
#include<iostream>
#include<conio.h>
using namespace std;
          class point
                    {
        int x,y;
        public:
        point(int abs=0, int ord=0)
                    {
           x=abs;y=ord;
          }
        int coincide(point*);
      };
                    int point::coincide(point* adpt)
                              {
          if ((adpt->x= =x)&&( adpt ->y= =y))return 1;
          else return 0;
        }
  main()
                    {
     point a, b(1),c(1,0);
     cout<<"a et b:" <<a.coincide(&b)<<"ou"<<b.coincide(&a)<<"\n";
```

```
      cout<<"b et c:" <<b.coincide(&c)<<"ou"<<c.coincide(&b)<<"\n";
getch() ;
return 0 ;
        }
```

2.4.2 Transmission par référence

La transmission par référence permet de mettre en place la transmission par adresse sans avoir à prendre en compte sois même la gestion.
Dans ce cas l'on pourrait modifier l'exemple précédant.

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


           class point
                      {
           int x, y;
           public:
           point(int abs=0, int ord=0)
            {
                      x=abs; y=ord;
              }
           int coincide(point & pt);
          };
             int point::coincide(point &pt)
               {  if ((pt.x=x)&&(pt.y=y))return 1;
              else return 0;
         }
 main()
                      {
      point a, b(1),c(1,0);
      cout<<"a et b:" <<a.coincide(b)<<"ou"<<b.coincide(a)<<"\n";
       cout<<"b et c:" <<b.coincide(c)<<"ou"<<c.coincide(b)<<"\n";
getch() ;
return 0 ;
  }
```

Remarque :
La transmission par valeur pose des problèmes qui se ramènent généralement à l'affectation et à l'initialisation.

2.5 Autoréférence

Pour l'autoréférence, on utilise le mot clé **this**. Il est utilisable uniquement au sein d'une fonction membre. Il désigne un pointeur sur l'objet appelé.

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

```
                    {
        int x,y;
        public:
        point(int abs=0, int ord=0)
         {
                    x=abs; y=ord;
        }
void affiche() ;
        } ;
void point ::affiche()
        {cout<< '' adresse ''<< this<< ''coordonnees''<<x<<'' ''<<y<<''\n'' ;      }
main()
{
        point a(5), b(3,15) ;
        a.affiche() ;
        b.affiche() ;
 getch() ;
return 0 ;
}
```

2.2.7. Fonctions membres statiques et membres constantes

Une fonction membre statique peut être appelée lorsqu'elle n'appartient à aucun objet de sa classe. Le mot réservé utilisé est **static**. L'utilisateur dans le cas des fonctions membres constantes précises les fonctions opérant sur les objets constants. Dans ce cas, la syntaxe utilisée est **const**.

Exemple :

Dans cet exemple on suppose qu'un point est constant

```
class point
        {
          int x, y ;
         public :
        point() ;
        void affiche const ;
        void deplace(..) ;
        } ;
```

On suppose la déclaration des objets suivants :

```
point  a ;
const point c ;
```

Seuls les objets déclarés **const** seront manipulés par affiche.

2.2.1. Pointeurs sur les fonctions membres

L'on peut utiliser les pointeurs sur les fonctions membres par exemple supposant que **point** ait deux fonctions membres de prototypes suivants :

```
        void dep_hor(int) ;
```

```
void dep_ver(int) ;
```

L'on peut déclarer:

```
void(point::*adf)(int);
```

Cette déclaration précise que adf est un pointeur sur une fonction membre de la classe point recevant un argument de type **int** et ne renvoyant aucune valeur.

Les affections suivantes seront alors valables :

```
adf =point :: dep_hor ; // ou adf =& point :: dep_hor ;
adf = point :: dep_hor;
```

Enfin si a est un objet de type **point,** l'on peut écrire (a.*adf)(3) ;

Ceci provoquera pour le point a, l'appel de la fonction membre dont l'adresse est contenue dans adf, en lui transmettant en argument la valeur 3.

2.6 Entrées / Sorties

2.6.1 Flots

Les entrées/sorties (clavier, écran, fichiers, périphériques) sont traitées comme des flots :

- Canal recevant de l'information (flot d'entrée) ;
- Canal émettant de l'information (flot de sortie).

4 flots sont prédéfinis :

- cin : entrée standard (en général le clavier) ;
- cout : sortie standard (en général l'écran) ;
- cerr : sortie standard d'erreur non bufférisée ;
- cerr : sortie standard d'erreur bufférisée ;

```
void main ()
{
    int i ;
    cin >> i ;                  //saisie d'un entier au clavier
    cout << '' i='' <<i ;       //à l'écran : i= …
}
```

2.6.2 Formatage des informations dans un flot en C

En C, le formatage doit apparaître dans chaque opération d'entrée-sortie :

```
void main ()
{
```

```
        float = 10.12345 ;
        printf ("%6.2f", x);      //à l'écran : 10.12
    }
```

En C++:

- On peut ne pas formater (on aura le formatage par défaut) ;
- On peut formater une seule fois pour toute une application.

2.6.3 Formatage des informations dans un flot en C++

class ios

Bits de formatage:

- skipws = 1: saut des espaces blancs (en entrée);
- left, right, internal = 1: cadrage;
- dec, oct, hex = 1: conversion décimale, octale, hexadécimale;
- unitbuf = 1: vide les tampons après chaque écriture.

2.6.4 Modification des bits de formatage

Manipulateurs non paramétriques : *Exp. : cout<<hex ;*

- ws active le saut des espaces blancs (entrée) ;
- dec, oct, hex converti en décimale, octale, hexadécimale (entrée/sortie) ;
- endl, ends : insère un saut de ligne et vide le tampon, insère une fin de chaîne (sortie) ;
- flush : vide le tampon (sortie).

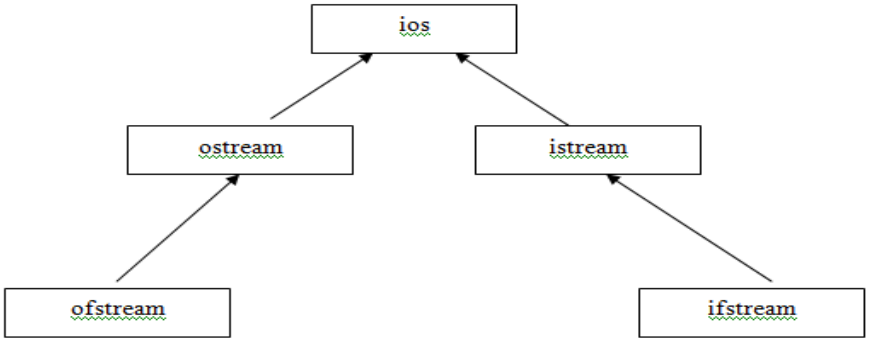
Manipulateurs paramétriques (#include <iomanip.h>) :

- setfill (int) : définit les caractères de remplissage ;
- resetiosflags (long), setiosflags (long) : modifie la valeur des bits de formatage ;
- setbase (int) : base de conversion (entrée/sortie) ;
- setprecision (int) : définit la précision des nombres flottants ;
- setw (int) : définit la précision.

Fonctions membres de la classe ios : *Exp. : cout.setf(ios ::oct)*

2.6.5 Accès à un fichier

Les bibliothèques utilisées :



2.6.6 Mots-clés en C++

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

3 Création et initialisation des objets : constructeurs et destructeurs

3.1 Notion de constructeur et de destructeur

3.1.1 Introduction au constructeur

Le constructeur est une fonction membre (définie comme les autres fonctions membres).

Il sera appelé automatiquement à chaque création d'un objet.

Un constructeur est une fonction membre systématiquement exécutée lors de la déclaration d'un objet statique, dynamique ou automatique.

Dans le cas de notre cours, nous traiterons le cas des objets automatiques.

Exemple :

```
#include<iostream>
#include<conio.h>
using namespace std;
class point
  {
   int x, y;
      public:
      point(); //constructeur de point
 void deplace(int, int);
void affiche();
};
point:: point()// initialisation par défaut grâce au constructeur
{
  x=20; y=20;
}
void point:: deplace(int dx,int dy)
{
 x+=dx;
 y+=dy;
}
void point::affiche()
{
cout<<"je suis en"<<x<<"et"<<y<<endl;
```

```
}
void tempo(int duree)
{
  float stop;
stop=duree+10000.0;
for( ;stop>0;stop=stop-10);
}
main()
{
point a, b;// les 2 objets initialisés en 20,10
a.affiche();
tempo(10);
a.deplace(17,10);
a.affiche();
tempo(15);
//clrscr();
b.affiche();
getch();
return 0;
}
```

Remarque : on remarque que le constructeur a la même **appellation** que la classe mais pour son utilisation, on n'utilise pas le **void**.

Lors de l'instanciation d'une classe définie par l'utilisateur, le compilateur effectue automatiquement la réservation de mémoire nécessaire aux fonctions membres de la classe considérée.
Cela se fait à l'aide des fonctions **malloc, calloc, free** dans la programmation en C pour permettre la réservation ensuite l'initialisation.
Le C++ permet à l'utilisateur de définir un constructeur, qui est appelé lors de l'instanciation après que l'espace pour les membres ait été réservé avec succès.
Le constructeur est une méthode particulière qui permet d'initialiser une instance de la classe considérée, et permet également d'atteindre l'abstraction souhaitée.
Généralement on définira le constructeur par défaut (**default constructor**), le constructeur d'assignation (**assigment constructor**) et le constructeur de copie (**copy constructor**).
Exemple de déclaration :

point z ; // constructeur par défaut

point zz(1.0,3.7); // constructeur d'assignation

// zz.re=1.0 ; zz.im=3.7 ;

point ze(zz) ; // constructeur de copie

// ze.re=zz.re ;

// ze.im=zz.im ;

Comme autre exemple l'on peut écrire :

point : : point(const point &) ; // constructeur de copie

point : : point() ; // constructeur par défaut

Exemple : utilisation des trois types de constructeurs

```
#include <iostream>
#include <conio.h>
using namespace std;
class vecteur
{
   float x;
   float y;
public:
            vecteur()
            {x=y=0;} //constructeur par défaut
   vecteur(float abs, float ord)
   {x=abs; y=ord;} // constructeur d'assignation
   vecteur (const vecteur &v) // constructeur de copie
   {x=v.x; y=v.y;}
   void affiche();
};
void vecteur::affiche()
   {cout <<"("<<x<<","<<y<<")"<<endl;}
int main()
{
```

```
    vecteur nul;// constructeur par défaut
            vecteur u1(1,2), u2(1,-1); // constructeur d'assignation
            vecteur u3(u2); //constructeur de copie
    nul.affiche();
    u1.affiche();
    u2.affiche();
    u3.affiche();
    return 0;
}
```

Remarque : Le constructeur par défaut ne fait que réserver de la place mémoire alors que le constructeur de copie en plus de réserver de la place, copie les membres un à un d'une instance source vers l'instance nouvellement créée.

Il est à remarquer que les implémentations par défaut sont dangereuses.

3.1.2 Notion de destructeur

Un objet peut posséder un destructeur, il s'agit également d'une fonction membre qui est appelée automatiquement au moment de la destruction de l'objet correspondant.

Dans le cas des objets automatiques, la destruction de l'objet a lieu lorsque l'on quitte le bloc où la fonction a été déclarée.

Par convention, le destructeur porte le même nom de la classe précédée du symbole tilde (~).

Le destructeur est une fonction membre systématiquement exécutée à la fin de la vie d'un objet statique, automatique ou dynamique.

Nous ne traiterons que des objets automatiques.

Remarque : on ne peut pas passer de paramètre par le destructeur

Exemple : construction et destruction d'objets

```
#include<iostream>
#include<conio.h>
using namespace std;
 class Test
{
  public:
```

```
      Test()
     {
          cout<<"\n Constructeur executee";
     }
      ~Test()
       {
          cout<<"\n Destructeur executee"<<endl;
       }
};
main()
{
    Test t;
    getch();
     return 0;
}
```

Exemple 2

```
#include<iostream>
#include<conio.h>
using namespace std;
class  test
  {    int num;
   public:
   test(int);// declaration constructeur
  ~test(); //  declaration destructeur
    };
```

```
test::test (int n) // definition constructeur
  {
 num=n;
 cout<<"appel constructeur num ="<<num<<endl;
  }
test::~test() // definition destructeur
  {
   cout<<"appel destructeur num= "<<num<<endl;
  }
main()
{
 void fonction (int);
 test a(1);
 for( int i=1;i<=2;i++)
 fonction(i);
getch() ;
return 0 ;
}
 void fonction(int p)
 {
  test n(2*p);
//  ~test n(2*p);
}
```

Il est nécessaire de définir par correspondance au constructeur un destructeur qui sera appelé en fin de vie de l'instance et qui se chargera de libérer proprement les ressources qui ont été allouées pour les membres de la classe lors de la durée de vie de l'objet.

1.1. Allocation dynamique

Lorsque les membres donnés d'une classe sont des pointeurs, on utilise aisément l'allocation dynamique.

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
Using namespace std;
class  calcul
  {
   int nbval,*val;
    public:
   calcul (int, int); // constructeur
  ~calcul(); // destructeur
void affiche();
};
calcul:: calcul(int nb, int nul)// constructeur
 {
  int i;
  nbval=nb;
  val=new int[nbval]; // réserve de la place
   for(i=0; i<nbval; i ++)
  val[i]=i*nul;
 }
calcul::~calcul() // destructeur
  {
   delete val; // abandon de la place réservée
  }
 void calcul::affiche()
  {
   int i;
   for(i=0; i<nbval; i++)
   cout<<val[i]<<endl;
  }
main()
{
 //clrscr();
 calcul suite1(10,4);
```

```
 suite1.affiche();
 calcul suite2(6,8);
 suite2.affiche();
  getch();
  return 0;
}
```

Exercice : écrire un programme permettant de faire de l'allocation dynamique à l'aide des opérateurs new et delete. On réservera de la place pour un entier, de 25 cases mémoires pour une chaîne de caractères sachant bien qu'un utilisateur utilisera un entier et une phrase.

```
#include<iostream.h>
#include<conio.h>
using namespace std;
main()
{
int *ad=new int;
char*adc;
adc=new char[25];
cout<<"entrez un nombre: " ;
cin>>*ad;
cout<<"voici le nombre:"<<*ad<<endl;
cout<<entrez une phrase:";
cin>>adc;// gets() précédé de fflush(stdin);
cout<<"voici la phrase :"<<adc<<endl;
delete ad;
delete adc;
getch();
return 0;
}
```

Remarque :

- Les opérateurs new et delete remplacent malloc et free (que l'on peut toujours utiliser) pour permettre de réserver et de libérer de l'espace.
- Il ne faut jamais utiliser conjointement malloc et delete ou bien new et free
- En C++, l'opérateur new permet de réserver au plus 64 ko de mémoire.

3.2 Propriétés

3.2.1 Objets automatiques et statiques

Les 2 types d'objets diffèrent par leur durée de vie (création et destruction) et les appels (constructeur et destructeur).

Objets automatiques

Ils sont créés par une déclaration soit dans une fonction, soit dans un bloc.

Dans une fonction, un objet est créé lors de sa déclaration puis détruit à la fin de l'exécution de la fonction et dans un bloc, cela se passe de la même manière que dans le cas de la fonction.

Objets statiques

Ils sont créés dans une déclaration située soit en dehors de toute fonction, soit dans une fonction mais assortie du qualificatif **static**.

Les objets statiques sont créés avant le début de la fonction main() et sont détruits après la fin de son exécution.

En résumé, un constructeur est appelé après la création de l'objet et le destructeur est appelé avant la destruction de l'objet.

Exemple **:**

```
#include<iostream>
#include<conio.h>
using namespace std;
 class point
{
  int x,y;
  public:
  point (int abs; int ord) // constructeur inline
      {
       x=abs, y=ord;
       cout<<"construction du point: ‘’<<x<<" "<<y<<endl;
      }
```

```
~point() // destructeur inline
   {
     cout<<" destruction du point : "<<x<<y<<endl;
  }
};
point a(1,1); // un objet statique de classe point
main()
{
 cout<<"debut programme<<endl;
   point b(10,10); // un objet automatique de class point
int i;
for(i=1;i<=3;i++)
{
  cout<<"boucle pour numéro : "<<i<<endl;
        point b(i,2*i); // objets crées dans un bloc
}
cout<<"Fin du programme ‘’<<endl;
getch() ;
return 0 ;
}
```

3.2.2 Objets dynamiques

L'allocation dynamique de l'emplacement mémoire requis est utilisée avec l'opérateur new qui appellera un constructeur de l'objet ; ce constructeur sera déterminé par la nature des arguments qui figurent à la suite de son appel.

La libération de l'emplacement mémoire correspondant est utilisée avec l'opérateur delete qui appellera le destructeur.

Exercice : utiliser new et delete dans l'exercice point dans le but de libérer ou d'allouer de l'espace mémoire.

3.2.3 Initialisation d'un objet lors de sa déclaration

D'une manière générale, lorsqu'on déclare un objet avec un initialiseur, ce dernier peut-être une expression d'un type quelconque, à condition qu'il existe un constructeur à un argument de ce type.

Pour initialiser un objet, l'on peut écrire :

point b=a ; // on initialise b avec l'objet a de même type

Manifestement, l'on aurait pu obtenir le même résultat en utilisant un constructeur par recopie ; c'est à dire :

point b(a) ; // on crée l'objet b en utilisant le constructeur par recopie de la classe point
// auquel on transmet l'objet a

En C++, pour l'initialisation, il est plus facile et plus précis d'écrire point a(1,2) ;

3.2.4 Tableaux d'objets

Le tableau d'objets n'est pas un objet. Dans l'esprit de la POO pure, ce concept n'existe pas puisque l'on manipule les objets. Mais l'on peut définir une classe dont un membre est un tableau d'objets.

Exemple :

```
class courbe
    {
     point p[20] ; // une courbe est un ensemble de 20 points
      ……..
   }
```

Dans le cas des tableaux d'objets, on peut également utiliser des constructeurs et des destructeurs.

```
#include<iostream>
#include<conio.h>
using namespace std;
 class point
  { int x, y;
    public:
    point (int abs=0, int ord=0)
      {
```

```
        x=abs; y=ord;
              cout<<" constructeur du centre "<<x<<"  "<<y<<endl;
             }
};
main()
{
 int n=3;
 point courbe[5]=(7,n,2*n+5);
}
```

3.2.5 Objets temporaires

A l'image des autres objets, il existe des objets dits temporaires et ces objets peuvent être utilisés dans un programme

```
#include<iostream.h>
#include<conio.h>
 class point
  { int x, y;
    public:
    point (int abs, int ord)
      {
        x= abs; y= ord;
              cout<<" constructeur du point "<<x<<"  "<<y<< '' a l'adresse :''<<this<<endl;
}
             ~point()
{
             cout<<''destruction du point ''<<x<<'' ''<<y<< '' a l'adresse :''<<this<<endl;
}
};
main()
             {
 point a(0,0) ; // objet automatique de classe point
a = point(1,2) ; // objet temporaire
```

```
a = point(3,5) ; // centre objet temporaire
cout<<'' fin du programme''<<endl ;
getch() ;
return 0 ;
}
```

4 Hiérarchie des classes : héritage

4.1 Introduction

L'héritage permet de donner à une classe toutes les caractéristiques d'une ou plusieurs autres classes. Les classes dont elle hérite sont appelées classes mères, classes de base ou classe antécédentes. La classe elle-même est appelée classe fille, classe dérivée ou classe descendante.

Les propriétés héritées sont les champs et les méthodes des classes de base.

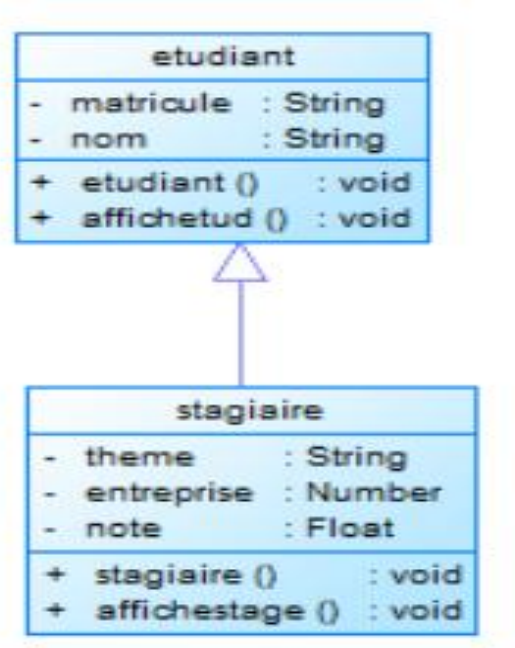
Le principe d'héritage est un fondement de la POO, il est à la base des possibilités de réutilisation des composants logiciels.

On crée une hiérarchie par imposition d'un ordre sur les abstractions. L'héritage est une hiérarchie fondamentale.

Également le Polymorphisme est un concept essentiel de la POO. C'est un moyen permettant de gérer les différences d'une collection d'abstraction en nous concentrant sur les points communs ; les racines grecques du terme poly (nombreux) et morphos (forme) indiquent bien que les valeurs d'une classe (collection des types dérivés d'un seul type ancêtre) et des opérations peuvent prendre de nombreuses formes.

4.2 Héritage simple

Soit une classe etudiant et stagiaire



Le programme ci-dessous décrit l'héritage simple décrit précédament.

```
#include <iostream>
#include <conio.h>

using namespace std;
class etudiant
{
	char* matricule;
	char* nom;
public:
	etudiant(char*, char*);
	void affichetud();
};
etudiant::etudiant(char* mat, char* n)
{
	matricule = mat;
	nom = n;
}
void etudiant::affichetud()
{
	cout << "Matricule : " << matricule << endl;
	cout << "Nom : " << nom << endl;
}
class stagiaire : public etudiant
{
	char* matricule;
	char* nom;
	char* theme;
	char* entreprise;
	float note;
public:
	stagiaire(char*, char*, char*, char*, float);
	void affichestage();
};
stagiaire::stagiaire(char* mat, char* n, char* th, char* ent, float N) : etudiant(mat,n)
```

```
{
        matricule = mat;
        nom = n;
        theme = th;
        entreprise = ent;
        note = N;
}
void stagiaire::affichestage()
{
        cout << "Matricule : " << matricule << endl;
        cout << "Nom : " << nom << endl;
        cout << "Theme : " << theme << endl;
        cout << "Entreprise : " << entreprise << endl;
        cout << "Note : " << note << endl;
}
main()
{
        etudiant kof("0012", "Koffi Ben");
        kof.affichetud();
        cout << endl;
        stagiaire coul("0031","Coul Karim","Conception SIG","BNET",14.5);
        coul.affichetud();
        cout << endl;
        coul.affichestage();
        cout << endl;
getch() ;
return 0 ;
}
```

Remarques :

- La classe stagiaire hérite de la classe étudiant.
- Il existe une relation d'héritage simple entre les classes étudiant (classe mère) et stagiaire (classe fille).
- On peut créer une classe etudiant.h que l'on va enregistrer dans le répertoire include (Voir TP)

- L'expression class stagiaire : public etudiant signifie que stagiaire est une classe dérivée de la classe de base etudiant et le mot public signifie que les membres de la classe de base (etudiant) seront des membres publics de la classe dérivée stagiaire.

A cet effet les objets de stagiaire seront déclarés ainsi :

stagiaire p, q;

Chaque objet de stagiaire peut faire appel :

- aux méthodes publiques de **stagiaire**
- aux méthodes publiques de **etudiant**

4.3 Objets membres

Il est possible qu'une classe possède un membre donné qui soit lui-même du type class.

A cet effet, les constructeurs et les destructeurs peuvent être utilisés.

Exemple :

```
#include<iostream>
#include<conio.h>
using namespace std;
 class point
  { int x, y;
    public:
    point (int abs=0, int ord=0)
      {
       x= abs; y=ord;
            cout<<" constructeur du centre "<<x<<"  "<<y<<endl;
      }
};
 class cercle
  {
    point centre; // class cercle possède un membre donné d'un type class
    int rayon;
    public:
      cercle(int,int,int);
  };
cercle::cercle(int abs, int ord, int ray): centre(abs,ord)
  {
```

```
   int rayon= ray;
   cout<<"construction du cercle de rayon ‘’<<rayon<<endl;
  }
main()
{
  cercle a(1,3,5);
  getch();
  return 0;
}
```

Exécution :

construction du centre 1 3

construction du cercle de rayon 5

les constructeurs sont appelés dans l’ordre suivant : point & cercle

s’il existe des destructeurs, il serait appelé dans l’ordre inverse.

4.4 Utilisation dans une classe dérivée des fonctions membres de la classe de base

Une classe dérivée n’a pas accès aux membres privés de sa classe de base.

Mais rien n’empêche à une classe d’accéder à n’importe quel membre public de sa classe de base.

```
#include <iostream>
using namespace std;
class etudiant
{
        char* matricule;
        char* nom;
public:
        etudiant(char*, char*);
        void affichetud();
};
etudiant::etudiant(char* mat, char* n)
{
        matricule = mat;
```

```
        nom = n;
}
void etudiant::affichetud()
{
        cout << "Matricule : " << matricule << endl;
        cout << "Nom : " << nom << endl;
}
class stagiaire : public etudiant
{
        char* matricule;
        char* nom;
        char* theme;
        char* entreprise;
        float note;
public:
        stagiaire(char*, char*, char*, char*, float);
        void affichestage();
};
stagiaire::stagiaire(char* mat, char* n, char* th, char* ent, float N) : etudiant(mat,n)
{
        matricule = mat;
        nom = n;
        theme = th;
        entreprise = ent;
        note = N;
}
void stagiaire::affichestage()
{
        etudiant::affichetud();
        cout << "Theme : " << theme << endl;
        cout << "Entreprise : " << entreprise << endl;
        cout << "Note : " << note << endl;
}
void main()
```

```
{
        etudiant kof("0012", "Koffi Ben");
        kof.affichetud();
        cout << endl;
        stagiaire coul("0031","Coul Karim","Conception SIG","BNET",14.5);
        coul.affichetud();
        cout << endl;
        coul.affichestage();
        cout << endl;
}
```

Ainsi l'on peut déclarer affichestage comme fonction faisant appel à affichetud de la classe etudiant.

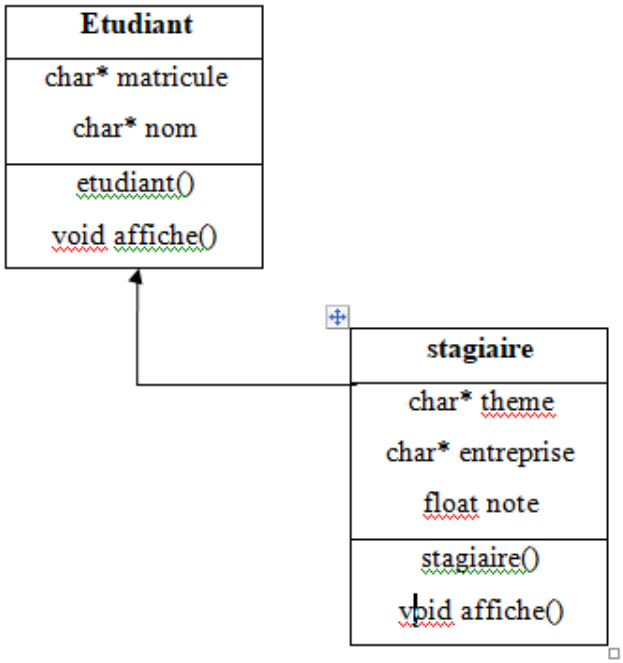
L'on a :

```
void stagiaire::affichestage()
{
        etudiant::affichetud();
        cout << "Theme : " << theme << endl;
        cout << "Entreprise : " << entreprise << endl;
        cout << "Note : " << note << endl;
}
```

Remarque : Cette technique a l'avantage d'utiliser les fonctions membres publiques de la classe de base puis de les enrichir selon la nécessité qui s'impose aux programmeurs.

4.5 Redéfinition de fonctions membres

Au lieu d'utiliser une fonction membre publique et l'enrichir dans une classe fille, l'on peut redéfinir tout simplement une fonction membre de la classe fille. Par exemple, l'on pourrait redéfinir la fonction affiche dans chacune des classes mais chacune ayant ses spécificités.



etudiant.h

```
#include <iostream>
using namespace std;
class etudiant
{
	char* matricule;
	char* nom;
public:
	etudiant(char*, char*);
	void affiche();
};
etudiant::etudiant(char* mat, char* n)
{
	matricule = mat;
	nom = n;
}

void etudiant::affiche()
{
	cout << "Matricule : " << matricule << endl;
```

```
        cout << "Nom : " << nom << endl;
}
```

main.cpp

```
#include <iostream>
using namespace std;
class stagiaire : public etudiant
{
        char* matricule;
        char* nom;
        char* theme;
        char* entreprise;
        float note;
public:
        stagiaire(char*, char*, char*, char*, float);
        void affiche();
};
stagiaire::stagiaire(char* mat, char* n, char* th, char* ent, float N) : etudiant(mat,n)
{
        matricule = mat;
        nom = n;
        theme = th;
        entreprise = ent;
        note = N;
}
void stagiaire::affiche()
{
        etudiant::affiche();
        cout << "Theme : " << theme << endl;
        cout << "Entreprise : " << entreprise << endl;
        cout << "Note : " << note << endl;
}
void main()
{
```

```
        etudiant kof("0012", "Koffi Ben");
        kof. etudiant::affiche();
        cout << endl;
        stagiaire coul("0031","Coul Karim","Conception SIG","BNET",14.5);
        coul. etudiant::affiche();
        cout << endl;
        coul.affiche();
        cout << endl;
}
```

4.6 Appel de constructeur et destructeur

Soit pointcouleur une classe qui hérite de point. Chacune des classes a un constructeur.

Les différents appels suivent une hiérarchie.

Par exemple pointcouleur a (10, 20, 5) entraîne :

- l'appel de point qui reçoit les arguments 10 & 20
- l'appel de pointcouleur reçoit les arguments 10,20 & 5.

Mais pointcouleur a (10,20) est rejeté par le compilateur.

Avec la méthode des arguments par défaut, l'on peut écrire :

pointcouleur (int abs=0, int ord=0, short cl=1) : point (abs, ord)

Dans ce cas, l'on peut aisément écrire :

pointcouleur b(5), c(3,2)

Par exemple l'appel de b(5) signifie :

- appel de point avec les arguments 5 & 0
- appel de pointcouleur avec les arguments 5,0 & 1.

Le code suivant récapitule ses appels de constructeurs.

```
#include<iostream.h>
#include<conio.h>
class  point
  {
   int x,y;
      public:
      point(int abs=0, int ord=0) //constructeur de point ( inline)
    {
```

```
      cout<<" constructeur du point"<<abs<<" "<<ord<<endl;
                        x=abs; y=ord;
    }
~point() //destructeur de point (inline)
           {
             cout<<"destruction du point:"<<x<<" "<<y<<endl;
      }
};
class pointcouleur: public point
  {
    short couleur;
    public:
    pointcouleur(int, int, short); // déclaration constructeur de pointcouleur
    ~pointcouleur() // déclaration destructeur de pointcouleur (inline)
     {
       cout<<"destruction de pointcouleur:"<<couleur<<endl;
     }
};
pointcouleur::pointcouleur(int abs=0;int ord=0; short cl=1): point(abs, ord)
    {
     cout<<"construction de pointcouleur: "<<abs<<" "<<ord<<" "<<cl<<endl;
          couleur=cl;
    }
main()  // programme principal
 {
  pointcouleur a(3,5,2), b(-1,3) ; // objet automatique
  pointcouleur c(9), d;
  pointcouleur *adr;
adr=new pointcouleur(18,25); //objet dynamique
 delete adr;
 getch();
 return 0;
}
```

4.7 Membres protégés

En plus des membres privés, publics, il y a ceux qui sont dits protégés.

Privé : Le membre n'est accessible qu'aux membres et fonctions amies de la classe.

Public : le membre est accessible non seulement aux fonctions membres et aux fonctions amies de la classe mais également à l'utilisateur de la classe mais, reste comparables à des membres publics pour le concepteur d'une classe dérivée.

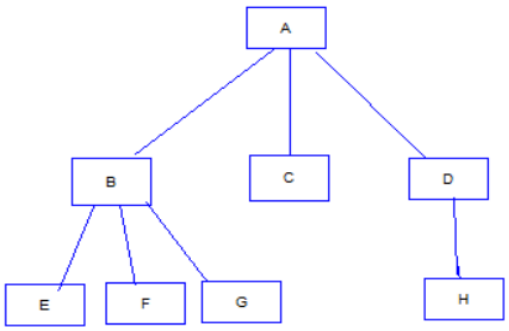
C'est un moyen de violer l'encapsulation des données. La syntaxe utilisée est **protected**.

Exemple :

```
class point
        {
          protected :
           int x,y ;
                public :
           point(…) ;
           void affiche() ;
           ……..
} ;
```

Résumé :

Nous venons d'aborder la notion d'héritage simple en mettant en évidence la notion de hiérarchie d'une classe de base à une classe unique fille. Cependant cet héritage simple peut-être complexe au regard de cet exemple.



Ici E est dérivé de B et B est dérivé de A.

On dit que :

B est un descendant direct de A (ou A est un ascendant direct de B)

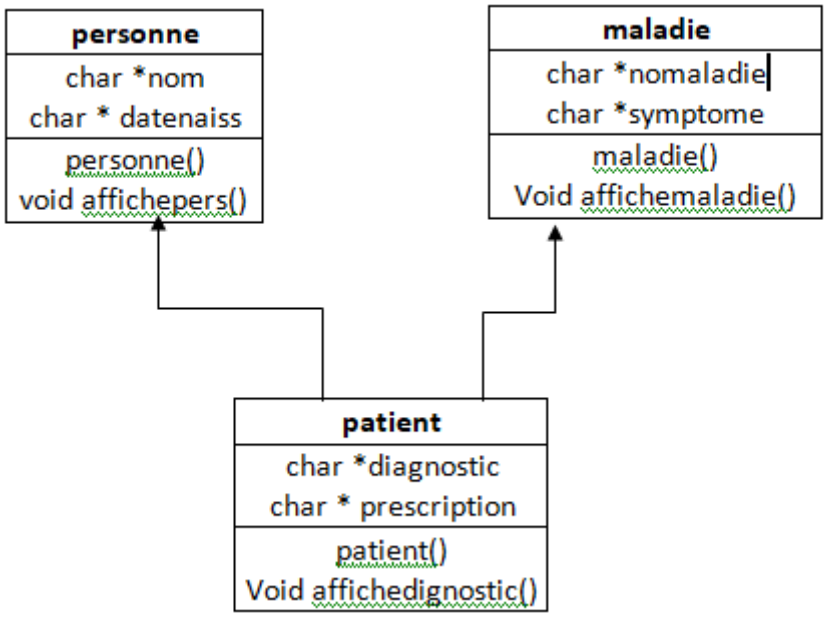
Et E est un descendant d'ordre 2 de A (ou A est un ascendant d'ordre 2 de E)

Outre cette possibilité, une classe peut hériter de plusieurs classes de base, c'est la notion d'héritage multiple.

4.8 Héritage multiple

Soit trois classes personne, maladie et patient. On suppose qu'un patient est une personne malade.

On schématise cela par un héritage multiple : la classe patient hérite des classes personne et maladie.



personne.h

```
#include <iostream>

using namespace std;

class personne

{

char* nom;

char* datenaiss;

public:

personne(char*,char*);

void affichepers();

};
```

```
personne::personne(char* n, char* date)

{

nom = n;

datenaiss = date;

}

void personne::affichepers()

{

cout << "Nom : " << nom << endl;

cout << "Date de naissance : " << datenaiss << endl;

}
```

maladie.h

```
#include <iostream>

using namespace std;

class maladie

{

char* nomaladie;

char* symptome;

public:

maladie(char*, char*);

void affichemaladie();

};

maladie::maladie(char* m, char* s)

{

nomaladie = m;

symptome = s;

}

void maladie::affichemaladie()
```

```
{
cout << "Nom de la maladie : " << nomaladie << endl;
cout << "Symptomes de la maladie : " << symptome << endl;
}
```

patient.h

```
#include <iostream>
#include <personne.h>
#include <maladie.h>
using namespace std;
class patient : public personne, public maladie
{
char* nom;
char* datenaiss;
char* nomaladie;
char* symptome;
char* diagnostic;
char* prescription;
public:
patient(char*, char*, char*, char*, char*, char*);
void affichediagnostic();
};
patient::patient(char* n, char* date, char* m, char* s, char* d, char* p) : personne(n, date), maladie(m, s)
{
nom = n;
datenaiss = date;
nomaladie = m;
symptome = s;
diagnostic = d;
prescription = p;
}
void patient::affichediagnostic()
{
```

```
personne::affichepers();
maladie::affichemaladie();
cout << "Diagnostic : " << diagnostic << endl;
cout << "Prescription : " << prescription << endl;
}
```

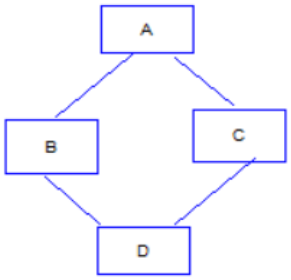
main.cpp

```
#include <iostream>
#include <patient.h>
using namespace std;
void main()
{
personne me("Kouadio Charles", "24/04/1991");
me.affichepers();
cout << endl;
maladie rhume("Rhume", "Migraines, nez coulant, mal de gorge");
rhume.affichemaladie();
cout << endl;
patient meRhume("Kouadio Charles", "24/04/1991", "Rhume", "Migraines, nez coulant, mal de gorge", "Paludisme Chronique", "Remontant anti-palu");
cout << "Information sur la personne" << endl;
meRhume.affichepers();
cout << endl;
cout << "Information sur la maladie" << endl;
meRhume.affichemaladie();
cout << endl;
cout << "Information sur le diagnostic" << endl;
meRhume.affichediagnostic();
cout << endl;
```

```
}
```

4.9 Classe virtuelle

Soit le schéma ci-dessous :



On a : class A { …….} ;

Class B : public A{…….};

Class C : public A{…….};

Class D : public B, public C

{…… } ;

On constate que D hérite 2 fois de A par l'intermédiaire de class B et C.

cela fait, appel à la notion de classe virtuelle. On incorpore une fois les membres de A dans D et on le précise dans la déclaration des classes B et C que A est virtuelle.

Le mot clé utilisé est **virtual** dans ce cas l'on écrit :

Class B : public virtual A{…….}

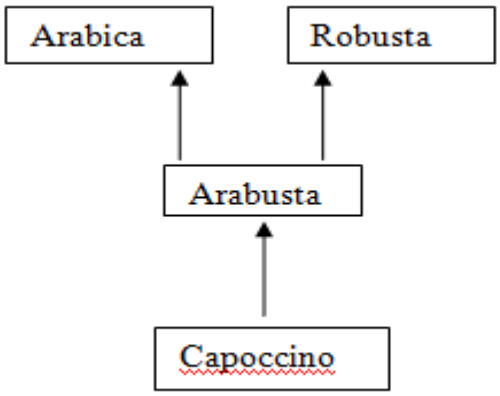
Class C : public virtual A{…….}

Class D : public B, public C

{…. } ;

Pour résoudre certains problèmes d'héritage, on déclare virtuelle la classe de base commune dans la donnée de l'héritage pour les classes filles. Les données de la classe de base ne seront alors plus dupliquées. Pour déclarer une classe mère comme une classe virtuelle, il faut faire précéder son nom du mot clé **virtual** dans l'héritage des classes filles.

Soit le schéma ci-dessous :



Arabica, robusta, et arabusta sont des races de café définies par les liens d'héritage ci-dessus. Capoccino est un produit fini d'arabusta.

Mettons en œuvre les notions d'héritage multiple, d'héritage simple et de classe virtuelle s'appuyant sur le schéma ci-dessus.

```
#include<iostream>
#include<conio.h>
#include<string.h>
using namespace std;
class arabica
{
 char* couleur;
 int taille;
  public:
   arabica(char* coul,int taye)
    {
      couleur=coul;
      taille=taye;
    } ;
   void affiche()
   {
     cout<<"couleur: "<<couleur<<endl;
```

```
       cout<<"taille: "<<taille<<"cm"<<endl;
    } ;
};
class robusta
{
  char* forme;
  int poids;
   public:
    robusta(char* fom,int poi)
     {
       forme=fom;
       poids=poi;
     };
  void affiche()
    {
       cout<<"forme: "<<forme<<endl;
        cout<<"poids: "<<poids<<" gramme"<<endl;
    } ;
};
class arabusta :public arabica , public robusta
{
     char* origine; char*nom;
     char* forme;
  int poids;
   char* couleur;
 int taille;
  public:
       arabusta(char* cou,int tay,char* frm,int pds,char* ori,char* name):arabica(cou,tay),robusta(frm,pds)
```

```
        {
            couleur=cou;
            taille=tay;
            forme=frm;
            poids=pds;
            origine=ori;
            nom=name;
                };
       void affiche()
        {
            arabica::affiche();
            robusta::affiche();
            cout<<"origine: "<<origine<<endl;
            cout<<"nom: "<<nom<<endl;
       } ;
};
class capoccino : virtual public arabusta
{
   int prix;
   int quantite;
    char* couleur;
   int taille;
    char* forme;
   int poids;
    char* origine; char*nom;
  public :
     capoccino(char* coule,int tai,char* fm,int pd,char* orig,char* nam,int pri,int
qte):arabusta(coule,tai,fm,pd,orig,nam)/*, arabica(coule,tai),robusta(fm,pd) */
```

```
        {
            prix=pri;
            quantite=qte;
            couleur=coule;
            taille=tai;
            forme=fm;
            poids=pd;
            origine=orig;
            nom=nam;
        };
       void affiche()
        {  arabusta::affiche();
            cout<<"prix: "<<prix<<"frs"<<endl;
            cout<<"quantite: "<<quantite<<"grammes"<<endl;
      } ;
  };
main()
{
   arabica ara("bleu",5) ;
   ara.arabica::affiche();
   cout<<endl;
   capoccino cap("rouge",5,"rond",10,"robusta","joli cafe",400,500);
   cap.capoccino::affiche();
   getch();
    }
```

4.10 Fonctions amies

Exemple :

```
#include<iostream.h>
```

```
#include<conio.h>
class  simple
 {
            int a,b;
            public:
               friend int somme(simple x);
               void initialise(inti,intj);
 };
void  simple::initialise_ab(inti,intj)
           {
        a=i;
             b=j;
           }
           main()
           {
            simple u;
            cout<<"la  somme de 3 et4vaut:"<<" ";
                        u.initialise_ab(3,4);
                        cout<<somme(u)<<endl;
                        getch();
                        return 0;
           }
```

4.4. S.T.L: Standard Template Library

Les fichiers d'en-tête

Algo.h, bool.h, bvector.h, degre.h ; function.h , list.h, nop.h, multiprap.h, multiset.h, Pair.h, random.c, set.h,

Exemple :

```
#include<iostream.h>
#include<conio.h>
#include<stadlib.h>
class  base
           {
```

```
            public:
            virtual int ajouter (int a, int b) { return( a+b ) };
            virtual int soustraire (int a, int b) { return(a-b)};
         virtual int multiplier(int a, int b) { return(a*b)};
       };
class  montremath: public base
          {
            virtual int multiplier(int a, int b)
                    {
                      cout<<"a*b:"<<endl;
                      return(a*b);
                    };
          };
class valeur_absolue:public base
     {
          virtual int  soustraire(int a, int b){ return (abs(a-b));}
     };
void  main(void)
          {
           Base=New Montremath;
           cout<<poly->ajouter(562,531)<<" "<<poly->soustraire(1500,407)<<endl;
           poly->multiplier(1093,1);
           poly=Newvaleur_absolue;
           cout<<poly->ajouter(892,20)<<" "poly->soustraire(0,1093)<<endl;
           cout<<poly->multiplier(1,1093);
           getch();
           return 0;
          }
```

5 Généricité ou les patrons

5.1 Introduction

Les modèles sont un outil très puissant de C++ qui ont été introduits relativement tard, avec la version CFRONT 3.0 du compilateur. L'utilité des modèles (appelés parfois **patrons** par certains auteurs), ou **classe paramétrable** par d'autres, ou **template** pour être bien compris, est sans doute pour certains auteurs comme un concept inutile mais cela n'est pas l'avis de l'auteur de ce cours. Ce modèle permet de définir une fonction ou une classe générique et de laisser le compilateur effectuer les ajouts nécessaires lors de l'implémentation définitive.

Un modèle est défini par le mot clé **template** suivi d'une liste.

La **liste de paramètres formels** est délimitée par une paire de signes d'inégalité (< ……..>) et les paramètres individuels sont séparés par des virgules.

La liste de paramètres formels ne doit pas en aucun cas être vide. Le mot template signifie **patron** en anglais.

5.2 Patrons de fonctions

La notion de surdéfinition de définition de fonction permet de donner à un nom unique plusieurs fonctions réalisant un travail différent.

La notion de patron est plus puissante et plus restrictive car il suffit d'écrire une seule fois la définition d'une fonction pour que le compilateur puisse automatiquement l'adopter à n'importe quel type. Elle est restrictive car par la nature même toutes les fonctions fabriquées par le compilateur devront répondre à la même définition donc au même algorithme.

5.2.1 Exemple de création et d'utilisation d'un patron de fonction

Les patrons de fonction sont aussi appelés les fonctions template. Un patron de fonction est une procédure ou une fonction sous la forme de template. La liste des paramètres formels est constituée d'une suite d'expressions constituées du mot réservé **class** suivi d'un identificateur.

Par exemple l'on peut définir une fonction paramétrée **minimum()** permettant de retourner le minimum de deux nombres.

```
template<class Type>
        Type minimum (Type, Type);
```

L'identificateur type remplace n'importe quel type de déclaration.

La liste de paramètres formels indique au compilateur que lors de l'instanciation du template, il devra remplacer l'identificateur type par le type d'instanciation.

Exemple :

```
#include<iostream>
```

```
#include<conio.h>

using namespace std;

template <class T>

 T minimum (T a, T b)

          {

            if(a<b) return a;

                    else return b;

          }

template <class T>

 T maximum (T a, T b)

          {

            return a>b?

a:b;

          }

main()

{

 int x=3; int y=5;

 cout<<"Min<int>(3,5)"<<minimum(x, y)<<endl;

 cout<<"Max<int>(3,5)"<<maximum(x, y)<<endl;

double a=-3.2; double b=0.0;

cout<<"Min<double>(-3.2,0.0)"<<minimum(a,b)<<endl;

cout<<"Max<double>(-3.2,0.0)"<<maximum(a,b)<<endl;

getch();

return 0;

}
```

5.2.2 Utilisation des patrons de fonctions

Pour les patrons, on peut utiliser n'importe quel type d'arguments tel que short, double, int*, char*, etc … ou des types définis comme des structures ou classes.

Exemple :

```
#include<iostream>
#include<conio.h>
#iclude<stdlib.h>
using namespace std;
template <class T>
 T minimum (T a, T b)
          {
            if(a<b) return a;
                    else return b;
        }
main()
{
 char*adr1="monsieur";
char*adr2="president";
 cout<<"min(adr1,adr2)="<< minimum (adr1,adr2);
getch();
return 0;
 }
```

On peut également faire un autre exemple en utilisant la même fonction générique minimum().

Exemple :

```
#include<iostream>
#include<conio.h>
using namespace std;
template <class T>
 T minimum (T a, T b)
          {
```

```
            if(a<b) return a;
                        else return b;
           }
class vecteur
{
 int x, y;
 public:
 vecteur(int abs=0;int ord=0)
  {
    x=abs; y=ord;
  }
void affiche()
 {
  cout<<" mes coordonnes :"<<x<<" "<< y<<endl;
 }
friend int operator < (vecteur, vecteur);
};
 int operator<(vecteur a, vecteur b);
{
  return a.x*a.x+a.y+a.y<b.x*b.x+b.y+b.y;
}
main()
 {vecteur u(-2,3),v(4,0),w;
w= minimum (u,v);
cout<<"min(u, v)="<<endl ;
w.affiche();
getch();
return 0;
```

```
}
```

5.2.3 Paramètres de type dans une définition d'un patron

Exemple :

```
Template <class T, class U>
  Fonction(T a, T*b, U c)
      {    ….. }
```

Les paramètres peuvent intervenir à n'importe quel endroit de la définition d'un patron. C'est à dire :

- ✓ Dans l'en-tête ;
- ✓ Dans la déclaration de variables locales ;
- ✓ Dans les instructions exécutables,

Par exemple l'on peut définir la fonction ci-dessus de la façon suivante

```
Template<class T, class U>
 Fonction(T  a, T*b, U c)
  {
   Tx ; // variable locale x de type T
    U*adr ; // variable locale de type U*
        …….
   Adr = new T[10];
    …..
   n=sizeof(T);
    …….
}
```

Chaque paramètre de type doit apparaître au moins une fois dans l'en-tête du patron.

5.2.4 Initialisation de variables

Dans l'utilisation de patrons de fonctions, il est possible d'initialiser des variables.

Exemple :

```
Template<class T>
 Fonction (T a)
   {
```

T X(5) ; // X est un objet local de type T qu'on construit en transmettant la valeur 5 a son constructeur

……..

}

5.2.5 Surdéfinition de patrons de fonction

Il est possible de surdéfinir un patron au même titre qu'une fonction normale, la seule condition étant que la signature de la fonction diffère.

Exemple 1:

```
#include<iostream>
#include<conio.h>
using namespace std;
template <class T>
 T minimum  (T a, T b) //patron 1
          {
            if(a<b) return a;
                    else return b;
          }
template <class T>
T minimum (T a, T b, T c )//patron 2
{
 return minimum (minimum (a, b),c);
}
main()
{
 int n=-5,p=7,q=3;
float x=5.2, y=9.15,z=0.5;
cout<< minimum (n , p)<<endl;// appel de patron 1
cout<< minimum (n, p, q)<<endl;// appel de patron 2
cout<< minimum (x, y, z)<<endl;// appel de patron 2
```

```
getch();

return 0;

}
```

Exemple 2:

```
#include<iostream.h>

#include<conio.h>

template <class T>

 T minimum (T a, T b)

          {

            if(a<b) return a;

                    else return b;

          }

template <class T>

 T minimum (T* a, T b) //patron 1

          {

            if(*a<b) return *a;

                    else return b;

          }

template <class T>

 T minimum (T a, T*b )

          {

            if(a<b*) return a;

                    else return b;

          }

template <class T>

 T minimum (T* a, T* b)

          {
```

```
            if(*a<b*) return a;
                        else return b;
          }
main()
{
 int n=12, p=-1;
float x=-2.5, y=5.0;
cout<< minimum (n, p)<<endl;//activation de int min(int, int)
cout<< minimum (&n, p)<<endl;//activation de int min(int*,int)
cout<< minimum (x,&y)<<endl;//activation de float min(float, float*)
cout<< minimum (&n,&p)<<endl;//activation de int min(int*,int*)
getch();
return 0;
}
```

Conclusion : un patron de fonction est une fonction dans laquelle le type de certains arguments est paramétré.

5.3 Patrons de classes

D'une manière analogue au patron de fonction, le C++ depuis la version 3 permet de définir les patrons de classe. Là encore, il suffit de décrire une seule fois la définition de la classe pour que le compilateur puisse automatiquement l'adapter aux différents types.

5.3.1 Création et utilisation d'un patron de classe

En s'inspirant de l'exemple classique point, l'on se rend compte que le type de coordonnées manipulées, est le type entier.

Cela impose à l'utilisateur de n'utiliser que des entiers. Un problème se pose, l'utilisateur pourrait être amené à utiliser des réels. Mais pour résoudre ce problème, l'on est amené à paramétrer la classe, c'est à dire créer un patron de classe. Et une fois le patron créé, l'on peut créer des instances de cette classe.

```
#include<iostream.h>
#include<conio.h>
```

```
template <class T>
 class point
{
 T x ; T  y;
 public:
  point (T abs=0, T ord=0)
  {
    x=abs; y=ord;
   }
 void affiche();
};
template<class T>
void point<T>::affiche()
{
 cout<<"Mes coordonnées sont: ‘’<<x<<"et"<<y<<endl;
}
main()
{
point<int> u(-3,5);
u.affiche();
point<char*> v("cos","sin");
v.affiche();
point<double> w(3.0,-1.5);
w.affiche();
getch();
return 0;
}
```

5.3.2 Paramètres de type d'un patron de classe

On peut utiliser à volonté les paramètres de type selon les besoins du développeur.

Supposons la classe suivante :

```
Template <class T, class U, class V> // 3 paramètres noms muets T, U, V
 class exemple
  {
    T n ; // un nombre de type T
    U t[5] ; // un tableau de 5 éléments de type U
    V fonction  (int U) ; // déclaration d'une fonction membre recevant 2 arguments de type
                          //    Int et U renvoyant un résultat de type V.
……
 }
```

En plus, on peut instancier la classe selon le besoin du programmeur.

Exemple : exemple<int, double, float>ex ;

Exemple<char*, int, double>ex1 ;

5.3.3 Paramètres d'expression d'un patron de classe

L'on peut utiliser des paramètres d'expression dans un patron de classe.

Pour une classe template < class T, int n>, l'on peut avoir 2 paramètres :

```
Class Tableau
 {
 …….
 } ;
```

- ✓ Le premier est un paramètre classique de type T
- ✓ Le second est un paramètre classique d'expression de type int.

```
#include<iostream.h>
#include<conio.h>
template <class T, int n>
 class tableau
{
```

```
 T tab[n];
 public:
  tableau()
   {
    cout<<"construction de tableau"<<endl;
   }
 T & operator[](int i)
   {
    return tab[i];
   }
};
class point
{
 int x, y;
 public:
 point( int abs=1, int ord=1)
   { x=abs; y=ord;
     cout<<"merci pour la construction du point"<<endl;
   }
  void affiche()
   {
    cout<<" mes coordonnées sont:"<<x<<"et"<<y<<endl;
   }
};
main()
{
 tableau <int,4> ti;
 int i;
 for(i=0;i<4;i++) ti[i]=i;
 cout<<"ti:";
```

```
        for(i=0;i<4;i++)
        cout<<ti[i]<<" ; "<<endl;
        tableau<point,3> tp;
         for(i=0;i<3;i++)
         tp[i].affiche();
         getch();
        return 0;
       }
```

5.3.4 Spécialisation d'un patron de classe

Un patron de classe définit une famille de classe dans laquelle chaque classe comporte à la fois sa définition et ses fonctions membres.

Ainsi, toutes les fonctions membres de noms donnés réalisent le même algorithme. Si l'on souhaite adapter une fonction membre à une situation particulière, il est possible d'en fournir une nouvelle.

```
#include<iostream>
#include<conio.h>
using namespace std;
template <class T>
 class point // création d'un patron de classe
  {
   T x; T y;
   public:
   point (T abs=  0, T ord=0)
    {
     x=abs; y=ord ;
    }
void affiche();
    };
// définition de la fonction affiche()
```

```
template<class T>

 void point<T>::affiche()

{

 cout<<"mes coordonnées sont :"<<(int)x<<"et"<<(int)y<<endl;

}

main()

{

point<int> a(-1,2);

a.affiche();

point<char*>b("compo 1","compo 2'');

b.affiche() ;

point<double> c(2.5,-1.2);

c.affiche();

getch();

return 0;

}
```

void point <char> :: affiche() précise au compilateur qu'il devrait utiliser cette fonction à la place de la fonction à la place de la fonction affiche du patron c'est à dire à la place l'instance point<char>.

Conclusion

Les classes **génériques** ou **patrons** ou **modèles** de classe permettent de définir des classes paramétrées par un type ou par une autre classe.

Ceci permet d'avoir une certaine généricité et est donc utile pour les structures de données.

6 Bibliographie

1. A laboratory for teaching Object-Oriented thinking, K. Beck, W. Cunningham, OOPSLA 1989
2. Aide-mémoire - C++. Christophe Pichaud. Broché. Dunod. 08/07/2020.
3. An Introduction to Object-Oriented Programming, T. Budd, Addison-Wesley 1991.
4. Apprendre la Programmation Orientée Objet avec le langage C#2ème édition. Luc Gervais. Manuel. Broché. Eni Editions. 05/09/2019.
5. BRAQUELAIRE Jean-Pierre, *Méthodologie de la programmation en C,* Masson, 1998.
6. C/C++ La bible du programmeur, Kris Jansa et Lars Klander, Les éditions Raynald Goulet Inc., 1999
7. C++ et Qt5Coffret 2 Volumes : Développez des applications professionnelles. Brice-Arnaud Guérin, Tristan Israel. Coffret. Eni Editions. 3/03/2019.
8. C++ primer. 2nd Edition., S.L. Lippman, Addison Wesley, 1992. En français : *L'essentiel du C++*. trad. par K. Zizi, Addison Wesley, 1992.
9. Charbonnel Jacquelin, *Langage C – les finesses d'un langage redoutable,* Armand Colin, 1992.
10. Classic Data Structures in C++, T.A. Budd, Addison-Wesley Publishing Company, 1994.
11. Comprendre et utiliser C++ pour programmer objets., G. Clavel, L. Trillaud, L. Veillon, Masson, Paris, 1994.
12. Exercices en langage C++175 exercices corrigés, 4ème édition. Claude Delannoy. Manuel. Broché. Eyrolles. 03/05/2018.
13. FEUER R. Alan, *Langage C – problèmes et exercices,* Masson, 1991.
14. Ingéniérie des objets, approche classe-relation, application à C++, P. Desfray, Masson, 1992.
15. Kernighan Brian, RITCHIE Denis, *Le langage C,* Dunod, 2000, (première édition 1978, première traduction 1990).
16. La programmation C et C++, D. Badouel, A. Khaled, Hermès 1993.
17. Langage C++De la programmation procédurale à l'objet, avec programmes d'illustration et exercices. Frédéric Drouillon. Manuel. Broché. Eni Editions. 12/09/2018.
18. Lapointe Nicolas, Pont entre C et C++, addison-wesley 1995.
19. Le guide du C++ moderne – De débutant à développeur. Mehdi Benharrats. , Benoît Vittupier. Broché. D-Booker. 22/10/2020.
20. Le langage C++, Bjarne Stroustrup Le créateur du C++ CampusPress, 2001
21. Les langages à objets, G. Masini, A. Napoli, D. Cobret, D. Léonard, K, Tombre, InterEditions, 1989
22. Modélisation par objets (la fin de la programmation), P. Desfray, Masson, 1996.
23. Object Oriented Design with Applications, Booch, G., Benjamin Cummings, 1991.
24. Object Oriented Software Engineering with C++, Darrel Ince, MacGraw-Hill 1991.
25. Object-Oriented Analysis, P. Coad, E. Yourdon, Yourdon Press Computing Series, 1991. Paru également en Français : Analyse Orientée Objet, Masson Prentice Hall,
26. Object-Oriented Modeling and Design, J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
27. Object-oriented Software Construction, B. Meyer, Prentice Hall, 1988. En français : *Conception et programmation par objets*. trad. par R. Mahl, InterEditions, 1991.
28. Object-Oriented Systems Analysis, Shlaer, Mellor, Yourdon Press Computing Series, 1988.
29. Programmer en C++ moderne De C++11 à C++20. Claude Delannoy. Broché. Eyrolles. 31/10/2019.
30. Programmer en langage C++ 3ed., C. Delannoy, Eyrolles, 1996.
31. Programmer en langage c++ 9 edition Couvre les versions c++11 c++14 et c++17 de la norme. DELANNOY CLAUDE.
32. Programmer en langage C Cours et exercices corrigés. Claude Delannoy. Format broché. Eyrolles. 01/07/2016.
33. Programmez avec le langage C++. Mathieu Nebra. Manuel. broché. Openclassrooms. 20/07/2015.

34. Software Engineering, I. Sommerville, Addison-Wesley, 1992. En français : *Le génie logiciel*. trad. par J.M. André, Addison-Wesley, 1992.
35. Stratégies et tactiques en C++, R.B. Murray, Addison-Wesley, 1994.
36. Stroustrup Bjarne, le langage C++, edition spéciale nouveauté, 2003.
37. The annotated C++ reference Manual, M.A. Ellis, B. Stroustrup, Addison-Wesley Publishing Company, 1990.
38. The C++ Programming Language 2ed., B. Stroustrup, Addison Wesley, 1992. En français : *Le langage C++*. trad. par H. Soulard, Addison-Wesley, 1992.
39. The Design and Evolution of C++, B. Stroustrup, Addison-Wesley, 1994.