Chapitre2

Programmation Orientée Objet (C++)

Plan

1- Introduction à la POO
2- Caractéristiques de base
3- Classes
4- Objets
5- Constructeur
6- Destructeur
7- Tableaux
8- Fonctions amies
9- Accesseur et Mutateur
10- Mot clé this
11- Membres statiques
12- Surcharge d'opérateurs
13-Classes imbriquées
14- Patron de fonction/classe
…

1. Introduction à la POO

La Programmation Orientée Objet (ou POO) est un paradigme de programmation dans lequel les programmes sont écrits et structurés autour des objets (et classes) plutôt que des fonctions.

1. Introduction à la POO

Classe et Objet:

Objet = Données + Traitements

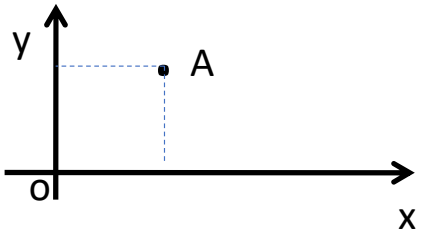
* **Classe**: Abstraction d'un ensemble d'objets qui ont les mêmes caractéristiques (attributs) et les mêmes comportements (méthodes)
* **Objet**: Instance d'une classe

La classe **Point**

Attributs
Abs_x
Ord_y
Méthodes
Dist_orig()

L'objet **A**

Attributs
2
3
Méthodes
$\sqrt{13}$



2. Caractéristiques de base

- **Encapsulation**: la protection de données
- **Héritage**: la transmission d'attributs et de méthodes aux classes dérivées
- **Polymorphisme**: la redéfinition d'une méthode
- **Multithreading**: l'exécution de plusieurs processus simultanément

3. Classes

- En C++, on sépare l'interface d'une classe et son corps.
- L'interface rassemble sous le nom de la classe ses attributs et les prototypes de ses méthodes (un fichier entête **.hpp**).
- Le corps de la classe contient le corps des méthodes définies dans l'interface de la classe (un fichier source **.cpp**).

3. Classes

- **Exemple**:

On considère une classe **Personne**

Personne
- string my_nom - string my_prenom
+ Personne(string prenom, string nom) + void quiSuisJe()

3. Classes

Interface de la classe Personne

```
// Personne.h
class Personne {
private:
    string my_prenom;
    string my_nom;
public:
    Personne( string prenom, string nom );
    void quiSuisJe();
};
```

Corps de la classe Personne

```
// Personne.cpp
// Constructeur
Personne::Personne( string prenom, string nom )
{
    my_prenom = prenom;
    my_nom = nom;
}
void Personne::quiSuisJe()
{
    cout << "Je suis " << my_prenom << " " << my_nom << endl;
}
```

3. Classes

- **Visibilité des membres**:
  - La visibilité des membres d'une classe (attributs et méthodes) est définie dans l'interface de la classe.
  - Trois mots-clés (**public**, **private**, **protected**) permettent de préciser l'accès aux membres.

3. Classes

- **public** : autorise l’accès pour tous.
- **private** : restreint l'accès aux seuls corps des méthodes de cette classe
- **protected** : autorise l'accès aux corps des méthodes des classes qui héritent de cette classe.

4. Objets

- Un objet est une instance d'une classe, il s'obtient en affectant des valeurs aux attributs.
- Un objet occupe un espace en mémoire, qui peut être alloué : **statiquement** ou **dynamiquement**.

4. Objets

- **Statiquement**:

On écrit le nom de la classe, suivi du nom de l'objet, suivi par les arguments d'appel donnés au constructeur de la classe.

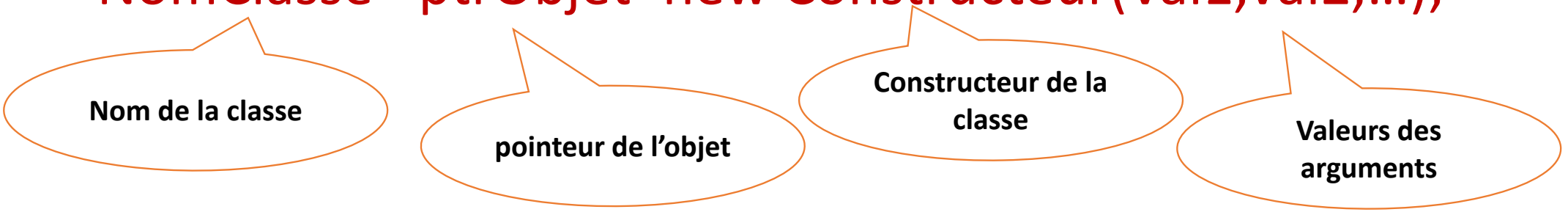
NomClasse NomObjet(Val1,Val2,…);

4. Objets

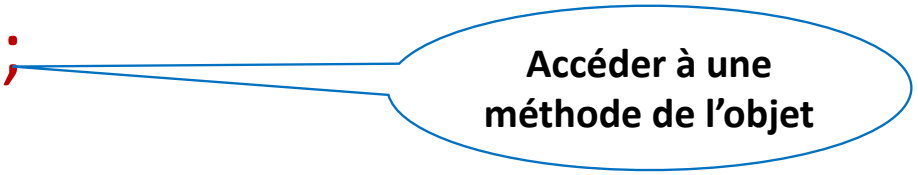
- **Dynamiquement**:
  - Ce type est réalisé via deux opérateurs(**new** et **delete**).
  - Un pointeur sur la zone mémoire où l'objet a été alloué est retourné.
  - A la fin du traitement, il désallouer.

4. Objets

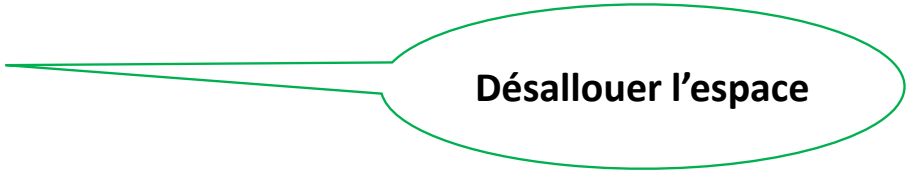
NomClasse* ptrObjet=new Constructeur(Val1,Val2,…);



ptrObjet -> méthode();



delete ptrObjet;



4. Objets

- **Accès aux attributs et méthodes**:

Etant donné une instance d'un objet, on accède à ses attributs et à ses méthodes grâce à la notation pointée (**.**), dans la limite de visibilité définie. Pour les pointeurs, cela se fait au travers de la notation fléchée (**->**).

4. Objets

- **Exemple**:
  - Etudiant etd("AAA","aaa","12345");
    etd.lire(); etd.moyenne();
  - Etudiant* ptretd=new Etudiant("AAA","aaa","12345");
    etd->lire(); etd->moyenne();

5. Constructeur

- Un constructeur est une méthode qui porte le même nom que la classe, il sert à initialiser un objet au moment de l'instanciation.
- Un constructeur par défaut ou un constructeur sans arguments est appelé au moment de l'instanciation d'un objet sans arguments d'appel.

5. Constructeur

- **Exemple**:

```
//Personne.hpp

class Personne{
    private:
        char *nom;
        char *prenom;
        int age;
    public:
        Personne();
        Personne(char*,char*,int);
        void lire();
        void afficher();
};
```

```
//Personne.cpp

Personne::Personne()
{

}
Personne::Personne(char *n,char *p,int a)
{
    nom=n;
    prenom=p;
    age=a;
}
void Personne::lire()
{
    cout<<"Donner le nom:"<<endl;
    cin>>nom;
    cout<<"Donner le prénom:"<<endl;
    cin>>prenom;
    cout<<"Donner l age:"<<endl;
    cin>>age;
}
```

5. Constructeur

- Un constructeur par copie permet l'instanciation d'un nouvel objet dans le même état qu'un objet existant.
- Il créé par défaut fait une initialisation champ à champ avec les champs de l'argument.

5. Constructeur

- Le constructeur copieur peut être appliqué en deux modes:
  - l'instanciation d'un objet avec en argument d'appel un objet du même type:

    Constructeur(NomClasse & NomObjet);
  - l'instanciation d'un objet et son initialisation avec un objet de la même classe: NomClasse V=E;

6. Destructeur

- Le destructeur est une méthode particulière qui est définie implicitement pour toutes les classes. Son nom est de la forme:

  ~NomClasse

- Il est appelé à la destruction/désallocation de l'objet.
- Il est indispensable lorsque l'on a besoin de faire de l'allocation dynamique.

6. Destructeur

- **Exemple**:

```
//Pile.hpp

class Pile{
    public:
        int *tab;
        int sommet;
    public:
        Pile(int);
        ~Pile();
};
```

Définition du destructeur

```
//Pile.cpp

Pile::Pile(int taille){
    tab=new int[taille];
    sommet=-1;
}

Pile::~Pile(){
    delete[] tab;
}
```

Déclaration du destructeur

Exercice 1

Rédiger la classe **Etudiant** définie comme suit:

Etudiant
- string nom //Nom de l'étudiant - String prénom //Prénom de l'étudiant - int cne // CNE de l'étudiant - Double [] notes //Notes des étudiants - int nb // Nombre des notes des étudiants

- Etudiant(string nm,string pr,int cn)
- void lire() // saisir les notes
- void moyenne() //calculer et afficher la moyenne des notes

Exercice 2

Rédiger la classe **NombresAmicaux** définie comme suit:

NombresAmicaux
- int a //Premier nombre - int b //Deuxième nombre

- NombresAmicaux(int ,int) //Initialiser les attributs
- int sommesDiviseurs(int) // Calculer et retourner la somme des diviseurs
- bool amicaux() //Retourner (true/false) selon la caractéristique des nombres amicaux
- void afficher(bool)//Afficher le résultat

Exercice 3

Rédiger la classe **Personne** définie comme suit:

Personne
- char nom[20] //Nom de la personne - char prenom[20] //Prénom de la personne - char cin[20]; // CIN de la personne

- Personne() //Constructeur par défaut
- Personne(char,char)// constructeur pour initialiser les attributs
- Personne(Personne &) // Constructeur par copie
- void lire() //saisir les attributs
- void afficher()//Afficher le résultat

Exercice 3

```hpp
//Personne.hpp
class Personne
{
    public:
        char nom[];
        char prenom[];
        char cinm[];
    public:
        Personne();
        Personne(char[],char[],char[]);
        Personne(Personne &);
        void lire();
        void afficher();

};
```

```cpp
//Personne.cpp
#include "Personne.hpp"
#include<iostream>
#include<cstring>
using namespace std;
Personne::Personne(){}
Personne::Personne(char nm[],char pr[],char cn[])
{
    strcpy(nom,nm);strcpy(prenom,pr);strcpy(cinm,cn);
}
Personne::Personne(Personne & p)
{
    strcpy(nom,p.nom);strcpy(prenom,p.prenom);strcpy(cinm,p.cinm);
}
void Personne::lire()
{
    cout<<"Entrer le nom: "; cin>>nom;
    cout<<endl;
    cout<<"Entrer le prenom: ";cin>>prenom;
    cout<<endl;
    cout<<"entrer cin: ";cin>>cinm;
}
void Personne::afficher()
{
    cout<<"nom: "<<nom<<" Prenom: "<<prenom<<" CIN: "<<cinm<<endl;
}
```

Exercice 3

```
int main()
{
    //Constructeur par défaut
    Personne p;
    p.lire();
    p.afficher();
    //Constructeur pour initialiser les attributs
    char nm[20],pr[20],cn[20];
    cout<<"Entrer Nom: ";cin>>nm;
    cout<<"Entrer Prenom: ";cin>>pr;
    cout<<"Entrer CIN: ";cin>>cn;
    Personne a(nm,pr,cn);
    a.afficher();
    //Constructeur par copie
    Personne b(a);
    b.afficher();
    return 0;
}
```

7. Tableaux

- Les tableaux en C++ sont des structures pouvant contenir un nombre fixe d'éléments de même nature. Chaque élément est accessible grâce à un indice correspondant à sa position dans le tableau:

```
Type nomTableau[taille];

Type nomTableau[taille][…]…;
```

7. Tableaux

- En déclarant un tableau, on dispose d'un pointeur (adresse du premier élément du tableau). Le nom d'un tableau est un pointeur sur le premier élément.
- La déclaration d'un tableau en utilisant un pointeur s'effectue comme suit:

```
Type *nomTableau = new Type[taille];
```

- Cette déclaration dynamique entraine l'utilisation de **delete** pour libérer l'espace mémoire

7. Tableaux

- **Exemple :**

```
#include<iostream>
using namespace std;
int main()
{
    int *tab,nb;
    tab=new int[50]; // Allocation dynamique
    cout<<"Entrer la taille du tableau: "<<endl;
    cin>>nb;
    cout<<"Lire un tableau:<<endl";
    for(int i=0;i<nb;i++)  // Lecture d'un tableau
    {
        cout<<"Donner elt n°: "<<i+1<<endl;
        cin>>tab[i];
    }
    cout<<"Afficher un tableau: "<<endl;
    for(int i=0;i<nb;i++)  // Affichage d'un tableau
    {
        cout<<" "<<tab[i];
    }
    cout<<endl;
    delete tab;  //Libération de l'espace mémoire
}
```

7. Tableaux

- Aussi, les tableaux dynamiques peuvent être déclarés à l'aide du type **vector** :

  vector <Type> nomTableau(taille);

- Le deuxième argument (**val**) ajouté dans la déclaration sert à initialiser toutes les cases du tableau:

  vector <Type> nomTableau(taille,val);

7. Tableaux

- L'ajout d'une case au tableau s'effectue à l'aide de la fonction **push_back()** :

```
nomTableau.push_back(valeur);
```

- La suppression d'une case du tableau se réalise à l'aide de la fonction **pop_back()**:

```
nomTableau.pop_back();
```

7. Tableaux

- **Exemple :**

```
#include<iostream>
#include <vector>
using namespace std;
int main()
{
    int nb;
    vector <int> tab(50); // Allocation dynamique
    cout<<"Entrer la taille du tableau: "<<endl;
    cin>>nb;
    cout<<"Lire un tableau:<<endl";
    for(int i=0;i<nb;i++)  // Lecture d'un tableau
    {
        cout<<"Donner elt n°: "<<i+1<<endl;
        cin>>tab[i];
    }
    cout<<"Afficher un tableau: "<<endl;
    for(int i=0;i<nb;i++)  // Affichage d'un tableau
    {
        cout<<" "<<tab[i];
    }
    cout<<endl;
}
```

7. Tableaux

- Un **tableau d'objets** est un ensemble d'éléments de même nature (NomClasse), où chaque élément un objet.
- Déclaration statique:

```
NomClasse nomTableau[taille];
```

- Déclaration dynamique:

```
NomClasse *ptr = new NomClasse[taille];
```

7. Tableaux

- **Exemple :**

```
#include<iostream>
using namespace std;
class Persons
{
   char name[30];
   int age;
   public:
      void getdata();
      void putdata();
};
```

7. Tableaux

- **Exemple :**

```
void Persons::putdata()
{
  cout<<"Nom: ";
  cin>>name;
  cout<<"Age: ";
  cin>>age;
}
void Persons::getdata()
{
  cout<< "Nom: " << name << "\n";
  cout<< "Age: " << age << "\n";
}
```

7. Tableaux

- **Exemple :**

```
int main ()
{ const int size=3 ;
  Persons persons[size] ;
  for(int i=0; i < size; i++)
  { cout<<"Entrez les informations sur la personne "<< (i+1) <<"\n";
    persons[i].putdata();
  }
  for(int i=0; i < size; i++)
  { cout<<"\nPersonne "<< (i+1) <<"\n";
    persons[i].getdata();
  }
  return 0;
}
```

Exercice 4

On considère la classe **Disque** définie comme suit:

Disque
- double rayon //Rayon du disque - double pi=3.14 //Constante pi

- Disque()
- Void lire();
- double surface(double) // Surface du disque
- double perimetre(double) //Périmètre du disque
- void affichage(double,double) //Affichage des résultats

Utiliser un tableau d'objets de type **Disque** pour manipuler un nombre d'objets de type **Disque**.

Exercice 4

```
//Disque.hpp
class Disque{
    public:
        double rayon;
        const double pi=3.14;
    public:
        Disque();
        void lire();
        double surface(double);
        double perimetre(double);
        void affichage(double,double);
};
```

```
#include"Disque.hpp"
#include<iostream>
#include<cmath>
using namespace std;
Disque::Disque(){
}
void Disque::lire()
{
    cout<<"Entrer le rayon:";
    cin>>rayon;
}
double Disque::surface(double r)
{
    double s;
    s=pi*pow(r,2);
    return s;
}
double Disque::perimetre(double r)
{
    double p;
    p=2*pi*r;
    return p;
}
void Disque::affichage(double s,double p)
{
    cout<<"Sirface: "<<s<<" et Perimetre: "<<p<<endl;
}
```

Exercice 4

```
int main()
{
    const int size=50 ;
    Disque tab[size] ;

    int nb;
    cout<<"Entrer le nombre de disques: "<<endl;
    cin>>nb;
    for(int i=0;i<nb;i++)
    {
        double r;
        cout<<"Disque n °: "<<i+1<<endl;
        tab[i].lire();

    }
    cout<<"Affichage des résultats: "<<endl;
    for(int i=0;i<nb;i++)
    {
        double s=tab[i].surface(tab[i].rayon);
        double p=tab[i].perimetre(tab[i].rayon);
        tab[i].affichage(s,p);
    }

    return 0;
}
```

8. Fonctions amies

- Il est possible d'accéder aux membres privés d'une classe à l'aide des fonctions amies.
- Les fonctions amies se déclarent en faisant précéder la déclaration classique de la fonction du mot clé ***friend*** à l'intérieur de la déclaration de la classe cible

8. Fonctions amies

- Il existe plusieurs situations d'amitié:
  - Une fonction indépendante est amie d'une ou de plusieurs classes.
  - Une ou plusieurs fonctions membres d'une classe sont amie d'une autre classe.

8. Fonctions amies

- **Fonction amie:**

```
class A
{
    int a; // Une donnée privée.
    friend void ecrit_a(int i); // Une fonction amie.
};
A essai;
void ecrit_a(int i)
{
    essai.a=i; // Initialise a.
    return;
}
```

8. Fonctions amies

- **Classe amie:**

```
class Hote
{    friend class Amie; // Toutes les méthodes de Amie sont amies.
     private:
        int i; // Donnée privée de la classe Hote.
     public:
        Hote(void)
        { i=0;
          return ;
        }
};
Hote h;
```

8. Fonctions amies

```
class Amie
{
     public:
        void print_hote(void)
        {
          printf("%d\n", h.i); // Accède à la donnée privée de h.
          return ;
        }
};
int main(void)
{
    Amie a;
    a.print_hote();
    return 0;
}
```

8. Fonctions amies

- **Remarques**:
  - une fonction amie d'une classe A amie d'une classe B n'est pas amie de la classe B, ni des classes dérivées de A
  - L'amitié **n'est pas transitive**. Cela signifie que les amis des amis ne sont pas des amis. Une classe A amie d'une classe B, elle-même amie d'une classe C, n'est pas amie de la classe C par défaut.
  - Les amis **ne sont pas hérités**. Ainsi, si une classe A est amie d'une classe B et que la classe C est une classe fille de la classe B, alors A n'est pas amie de la classe C par défaut.

Exercice 5

Soit une classe **vecteur3d** ayant 3 données membres privées, de type entier, les composantes du vecteur (x,y,z). Elle a un constructeur permettant d'initialiser les données membres.

Ecrire une fonction indépendante, **coincide**, amie de la classe **vecteur3d**, permettant de savoir si 2 vecteurs ont mêmes composantes.

Si v1 et v2 désignent 2 vecteurs de type vecteur3d, écrire le programme qui permet de tester l'égalité de ces 2 vecteurs.

Exercice 5

```hpp
//FonctionAmie.hpp
class vecteur3d{
    private:
        int x;
        int y;
        int z;
    public:
        vecteur3d(int a=0,int b=0, int c=0);
        friend void coincide (vecteur3d p, vecteur3d q);
};
```

```cpp
//FonctionAmie.cpp
#include"FonctionAmie.hpp"
#include <iostream>
using namespace std;

vecteur3d::vecteur3d(int a,int b,int c)
{
    x=a;
    y=b;
    z=c;
}
void coincide(vecteur3d p, vecteur3d q)
{
    if(p.x==q.x && p.y==q.y && p.z==q.z)
    {
        cout<<"Les 2 vecteurs sont égaux"<<endl;
    }
    else cout<<"Les 2 vecteurs sont différents"<<endl;
}
```

Exercice 5

```
int main()
{
vecteur3d v1(3,2,5);
vecteur3d v2(3,4,5);
coincide(v1,v2);

vecteur3d v3(3,2,5);
vecteur3d v4(3,2,5);
coincide(v3,v4);
return 0;
}
```

```
Les 2 vecteurs sont diffÚrents
Les 2 vecteurs sont Úgaux


--------------------------------
```

9. Accesseurs et Mutateurs

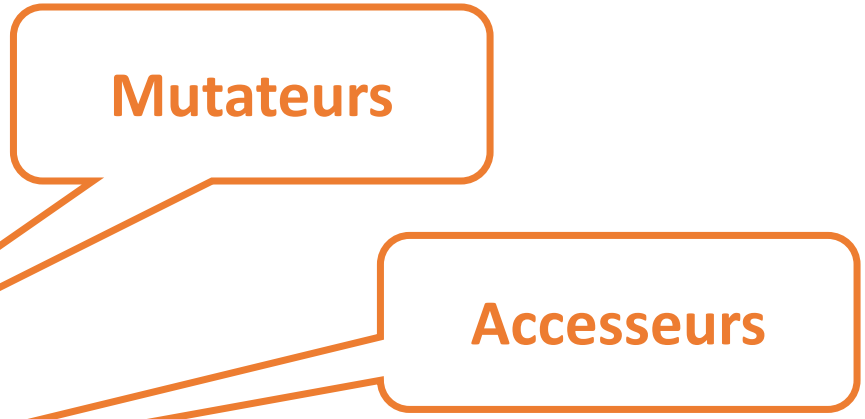
- L'accès aux données membres privées s'effectue à l'aide d'un ensemble de fonctions appelées: Accesseurs et Mutateurs.
- Les accesseurs et les mutateurs seront publiques.
- Les accesseurs sont des fonctions qui permettent de récupérer le contenu.
- Les mutateurs sont des fonctions qui permettent de modifier le contenu.

9. Accesseurs et Mutateurs

- **Exemple:**

```
//Point.hpp

class Point
{
    private:
        double x,y;
    public:
        void setX(double );
        void setY(double );
        double getX();
        double getY();
        double distance(const Point &);
        Point milieu(const Point &);
        void saisir();
        void afficher();
};
```



9. Accesseurs et Mutateurs

Mutateurs

Accesseurs

```
#include "Point.hpp"
#include <cmath>
#include <iostream>
using namespace std;
void Point::setX(double a)
{   x = a;  }
void Point::setY(double b)
{   y = b;  }
double Point::getX()
{   return x; }
double Point::getY()
{ return y; }
double Point::distance(const Point &P)
{  double dx, dy;
    dx = x - P.x;
    dy = y - P.y;
    return sqrt(dx*dx + dy*dy);
}
```

```
Point Point::milieu(const Point &P)
{   Point M;
    M.x = (P.x + x) /2;
    M.y = (P.y + y) /2;
    return M;
}
void Point::saisir()
{   cout << "Tapez l'abscisse : "; cin >> x;
    cout << "Tapez l'ordonnée : "; cin >> y;
}
void Point::afficher()
{   cout << "L'abscisse vaut " << x << endl;
    cout << "L'abscisse vaut " << y << endl;
}
```

9. Accesseurs et Mutateurs

```
int main()
{
Point A, B, C;
double d,abs,ord;
cout << "SAISIE DU POINT A" << endl;
A.saisir();
cout << endl;
cout << "SAISIE DU POINT B" << endl;
cout << "Tapez l'abscisse : "; cin >> abs;cout << endl;
cout << "Tapez l'ordonnée : "; cin >> ord;cout << endl;
B.setX(abs);
B.setY(ord);
if(A.getX()==B.getX() && A.getY()==B.getY())
        cout<<" Les deux points sont identiques"<<endl;
else{
        C = A.milieu(B);
        d = A.distance(B);
        cout << "MILIEU DE AB" << endl;
        C.afficher();
        cout << endl;
        cout << "La distance AB vaut :" << d << endl;
    }
return 0;
}
```

```
SAISIE DU POINT A
Tapez l'abscisse : 5
Tapez l'ordonnÚe : 7

SAISIE DU POINT B
Tapez l'abscisse : 3

Tapez l'ordonnÚe : 5

MILIEU DE AB
L'abscisse vaut 4
L'abscisse vaut 6

La distance AB vaut :2.82843
```

10. Mot clé this

- Le mot clé ***this*** permet de désigner l'objet dans lequel on se trouve.
- lorsque l'on désire accéder à une donnée membre d'un objet à partir d'une fonction membre du même objet, il suffit de faire précéder le nom de la donnée membre par ***this->***.

10. Mot clé this

- **Exemple:**

```
//Point.hpp

class Point
{
    private:
        double x,y;
    public:
        void setX(double );
        void setY(double );
        double getX();
        double getY();
        void saisir();
        void afficher();
};
```

Utilisation de this

```
#include "Point.hpp"
#include <cmath>
#include <iostream>
using namespace std;
void Point::setX(double x)
{   this->x = x;  }
void Point::setY(double y)
{   this->y = y;  }
double Point::getX()
{   return x; }
double Point::getY()
{ return y; }
void Point::saisir()
{   cout << "Tapez l'abscisse : "; cin >> x;
    cout << "Tapez l'ordonnée : "; cin >> y;
}
void Point::afficher()
{   cout << "L'abscisse vaut " << x << endl;
    cout << "L'l'ordonnée vaut " << y << endl;
}
```

10. Mot clé this

```
int main()
{
Point A;
int abs,ord;
cout << "SAISIE DU POINT A" << endl;
cout << "Tapez l'abscisse : "; cin >> abs;cout << endl;
cout << "Tapez l'ordonnée : "; cin >> ord;cout << endl;
A.setX(abs);
A.setY(ord);
A.afficher();
return 0;
}
```

```
SAISIE DU POINT A
Tapez l'abscisse : 5

Tapez l'ordonnÚe : 7

L'abscisse vaut 5
L'l'ordonnÚe vaut 7
```

11. Membres statiques

- Les membres (Attributs et Fonctions) d'une classe sont liés à chaque instance de la classe.
- Il est possible de lier un membre (Attribut ou Fonction) à la classe elle-même, et dans ce cas on parle d'un **membre statique**.
- Le membre statique est partagé par toutes les instances et la classe elle-même.
- Pour désigner un membre statique, il faut utiliser le mot clé ***static***.

11. Membres statiques

- **Exemple: (Attribut statique)**

```
//AttStatic.hpp

class AttStatic{
    public:
        static int compteur;
    public:
            AttStatic();
            void afficher();
};
```

Attribut statique

11. Membres statiques

- **Exemple: (Attribut statique)**

```
// AttStatic.cpp
#include"AttStatic.hpp"
#include<iostream>
using namespace std;
int AttStatic::compteur=0;
AttStatic::AttStatic()
{
    compteur++;
}
void AttStatic::afficher()
{
    cout<<" Compteur = "<<compteur<<endl;
}
```

Initialisation de l'attribut statique

11. Membres statiques

- **Exemple: (Attribut statique)**

```
int main()
{
    cout<<" Avant instanciation"<<endl;
    cout<<" Compteur = "<<AttStatic::compteur<<endl;
    cout<<" Après instanciation"<<endl;
    AttStatic A;
    A.afficher();
    AttStatic B;
    B.afficher();
    AttStatic C;
    C.afficher();
}
```

Utilisation de l'attribut statique

```
Avant instanciation
Compteur = 0
AprÞs instanciation
Compteur = 1
Compteur = 2
Compteur = 3
```

11. Membres statiques

- **Exemple: (fonction statique)**

```
//FonctStatic.hpp

class FonctStatic{
    public:
        static int compteur;
    public:
            FonctStatic();
            static void afficher();
};
```

Attribut statique

Fonction statique

11. Membres statiques

- **Exemple: (fonction statique)**

```
#include"FonctStatic.hpp"
#include<iostream>
using namespace std;
int FonctStatic::compteur=0;
FonctStatic::FonctStatic()
{
    compteur++;
}
void FonctStatic::afficher()
{
    cout<<" Compteur = "<<compteur<<endl;
}
```

Initialisation de l'attribut statique

Définition de la fonction statique

11. Membres statiques

- **Exemple: (fonction statique)**

```
int main()
{
    cout<<" Avant instanciation"<<endl;
    cout<<" Compteur = "<<FonctStatic::compteur<<endl;
    cout<<" Après instanciation"<<endl;
    FonctStatic A;
    FonctStatic::afficher();
    FonctStatic B;
    FonctStatic::afficher();
    FonctStatic C;
    FonctStatic::afficher();
}
```

Utilisation de l'attribut statique

Utilisation de la fonction statique

12. Surcharge d'opérateurs

- En C++, les opérateurs usuels (+,-,*, …) peuvent être **surchargés**.
- La **surcharge d'opérateurs** représente la possibilité de reprogrammer les opérateurs pour faire autre chose que ce qu'ils font d'habitude.
- La **surcharge d'un opérateur <op>** se fait en déclarant, au sein d'une classe, une fonction membre appelée **operator <op>**.

12. Surcharge d'opérateurs

- Plusieurs cas peuvent se présenter, selon que **<op>** est un opérateur: **Unaire** ou **Binaire**.
- **Opérateur unaire**: un seul argument
- **Opérateur binaire**: deux arguments

12. Surcharge d'opérateurs

- **Exemple: (Opérateur unaire)**

On considère qu'on a une classe **Complexe**, et on veut surcharger l'opérateur **-** pour qu'il calcule l'opposé d'un nombre complexe:

```
//Prototype
        Complexe operateur-( )
```

```
//Définition
        Complexe operateur-( )
{return Complexe(-re,-im);
}
```

12. Surcharge d'opérateurs

- **Exemple: (Opérateur binaire)**

On considère qu'on a une classe **Complexe**, et on veut surcharger l'opérateur **+** pour qu'il calcule la somme de deux nombres complexes:

```
//Prototype
  Complexe operateur+(Complexe)
```

```
//Définition
  Complexe operateur+(Complexe a)
{return Complexe(a.re+re;a.im+im);
}
```

Exercice 6

On considère la classe **SurchPoint** définie comme suit:

SurchPoint
- double x //Abscisse - double y //Ordonnée

- SurchPoint()
- SurchPoint(double,double)
- Void saisir_coord();
- SurchPoint operator+(SurchPoint) // Calculer la somme
- void affichage() //Affichage des résultats

Donner un programme pour tester la surcharge de l'opérateur +.

Exercice 6

```
//SurchPoint.hpp

class SurchPoint
{
    public:
        double x,y;
    public:
        SurchPoint();
        SurchPoint (double,double );
        void saisir_coord();
        SurchPoint operator+(SurchPoint);
        void afficher();
};
```

Déclaration de operator+

Exercice 6

```cpp
//SurchPoint.cpp
#include "SurchPoint.hpp"
#include <cmath>
#include <iostream>
using namespace std;
SurchPoint::SurchPoint() { }
SurchPoint::SurchPoint(double x,double y)
{
    this->x=x;
    this->y=y;
}
void SurchPoint::saisir_coord()
{
    cout<<" Entrer abscisse: ";  cin>>x;  cout<<endl;
    cout<<" Entrer ordonnée: ";   cin>>y;  cout<<endl;
}
SurchPoint SurchPoint::operator+(SurchPoint A)
{
    return SurchPoint(x+A.x,y+A.y);
}
void SurchPoint::afficher()
{
    cout<<" Les coordonnées du point sont: "<<endl;
    cout<<" Abscisse: "<<x<<endl;
    cout<<" Ordonnée: "<<y<<endl;
}
```

Définition de operator+

Exercice 6

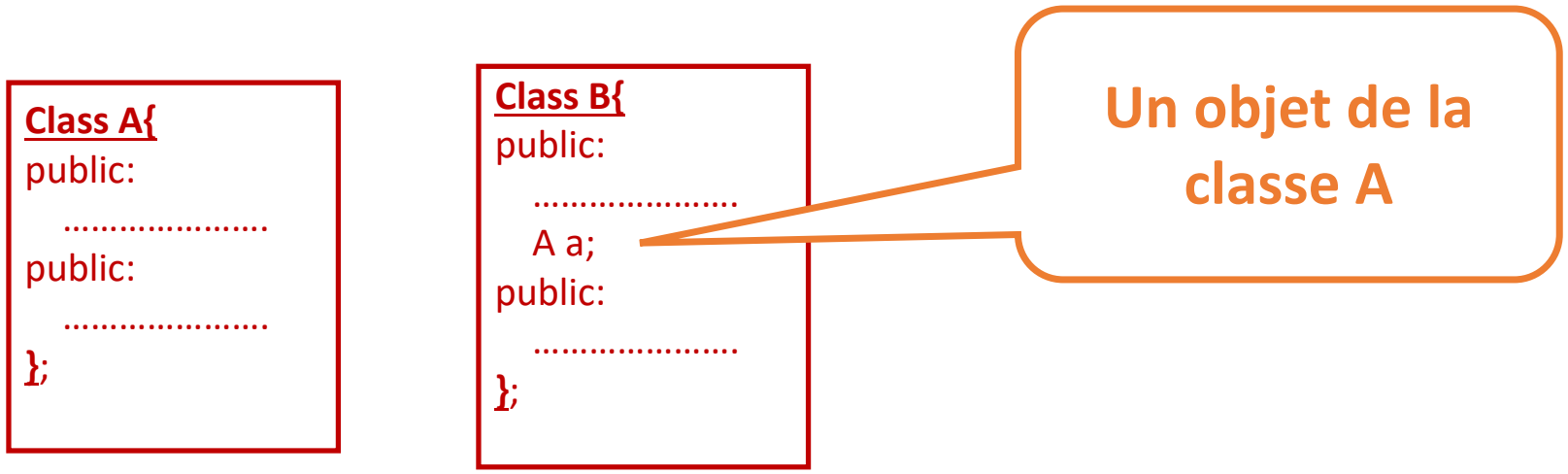
```
int main()
{    SurchPoint D;
     SurchPoint B(5,7);
     B.afficher();
     SurchPoint C(3,2);
     C.afficher();
     D=B.operator+(C);
     D.afficher();
     return 0;
}
```

Utilisation de operator+

```
Les coordonnÚes du point sont:
Abscisse: 5
OrdonnÚe: 7
Les coordonnÚes du point sont:
Abscisse: 3
OrdonnÚe: 2
Les coordonnÚes du point sont:
Abscisse: 8
OrdonnÚe: 9
```

13. Classes imbriquées

- La notion de classes imbriquées représente la possibilité d'utiliser une classe à l'intérieur d'une autre classe.



13. Classes imbriquées

- **Exemple:**

On considère deux classes: Etablissement et Etudiant:

```
//Etud.hpp
class Etud{
  public:
        int ord;
        char nom[20];
  public:
         Etud();
         void lire();
         …
}
```

```
//Etablissement.hpp
class Etablissement{
  public:
        int nb;
        Etud* liste;
  public:
         Etatblissement();
         void lire_liste();
         …
}
```

Exercice 7

On considère les classes **Etudiant** et **Etablissement** :

Etudiant
- int ord - char nom[30]
- Etudiant() - Void lire() - Void afficher();

Etablissement
- int nb - char nomEtab[30] - Etudiant *liste
- Etablissement() - Void lireInfo() - Void afficheInfo() - ~Etablissement()

Donner un programme pour tester les classes imbriquées

Exercice 7

```
//Etud.hpp
class Etud{
    public:
        int ord;
        char nom[20];
    public:
        Etud();
        void lire();
        void afficher();
};
```

```
//Etablissement.hpp"

class Etablissement{
    public:
        int nb;
        char nomEtab[30];
        Etud *liste;
    public:
        Etablissement(int const ,int);
        void lireInfo();
        void afficherInfo();
        ~Etablissement();//destructeur
};
```

Exercice 7

```
#include"Etudiant.hpp"
#include"Etablissement.hpp"
#include<iostream>
using namespace std;
Etud::Etud(){}
void Etud::lire()
{
    cout<<" Saisir les infos de l etudiant: "<<endl;
    cout<<" Ordre: ";  cin>>ord; cout<<endl;
    cout<<" Nom: ";  cin>>nom;  cout<<endl;
}
void Etud::afficher()
{
    cout<<" Afficher les infos de l etudiant: "<<endl;
    cout<<" Ordre: "<<ord<<endl;
    cout<<" Nom: "<<nom<<endl;
}
```

Exercice 7

```
Etablissement::Etablissement(int const max,int n){liste=new Etud[max];nb=n;}
void Etablissement::lireInfo()
{
    cout<<" Entrer le nom d etablissement: "; cin>>nomEtab;
    for(int i=0; i<nb;i++)
    {
        liste[i].lire();
    }
}
void Etablissement::afficherInfo()
{
    cout<<" Le nom d etablissement: "<<nomEtab<<endl;
    for(int i=0; i<nb;i++)
    {
        liste[i].afficher();
    }
}
Etablissement::~Etablissement()
{
    delete[] liste;
    cout<<" liberation de la memoire"<<endl;
}
```

Exercice 7

```
int main()
{
    const int mx=100;
    int nombre=3;
    Etablissement fst(mx,nombre);
    fst.lireInfo();
    fst.afficherInfo();

    return 0;
}
```

```
Entrer le nom d etablissement: EST_Beni_Mellal
Saisir les infos de l etudiant:
Ordre: 1

Nom: AAAAA

Saisir les infos de l etudiant:
Ordre: 2

Nom: BBBBB

Saisir les infos de l etudiant:
Ordre: 3

Nom: CCCCC

Le nom d etablissement: EST_Beni_Mellal
Afficher les infos de l etudiant:
Ordre: 1
Nom: AAAAA
Afficher les infos de l etudiant:
Ordre: 2
Nom: BBBBB
Afficher les infos de l etudiant:
Ordre: 3
Nom: CCCCC
liberation de la memoire
```

14. Patron de fonction/Classe

- Un **patron** est un **template** définissant un modèle de fonction ou de classe dont certaines parties sont des paramètres (type traité, taille maximale). Il sert à généraliser un même algorithme, une même méthode de traitement, à différents cas ou types de données.
- Le mot clé **template** est suivi de la liste des paramètres du patron entre les signes **<** et **>**, suivi de la définition de la fonction ou de la classe.

14. Patron de fonction/Classe

Patron de fonction:

- Un patron de fonction est un modèle de fonction précédé du mot clé **template** et de la liste des paramètres du patron entre les signes **<** et **>**.

```
Template<typename T> ….
        //Déclaration de la fonction en utilisant le type T
```

14. Patron de fonction/Classe

Exemple: (Intérêt des patrons des fonctions par rapport aux surdéfinitions des fonctions)

```
int min(const int a, const int b)
{
  return a < b ? a : b;
}

float min(const float a, const float b)
{
  return a < b ? a : b;
}

char min(const char a, const char b)
{
  return a < b ? a : b;
}
```

Surdéfinition de la fonction min()

14. Patron de fonction/Classe

```
template<typename T> T min(const T & a, const T & b)
{
  return a < b ? a : b;
}
```

Patron de la fonction min()

14. Patron de fonction/Classe

Patron de classe:

- La définition d'un patron de classe est similaire à la définition du patron de fonction, c'est-à dire le mot clé **template** est utilisé ainsi que les deux signes **<** et **>** pour désigner les paramètres.

```
Template<typename T> class ….
        //Déclaration de la classe en utilisant le type T
```

14. Patron de fonction/Classe

Exemple:

```
template<typename T> class Point
{
    private:
        T m_x;
        T m_y;
    public:
        Point(T abs = 0, T ord = 0);
        void affiche();

};
```

Patron de la classe Point

14. Patron de fonction/Classe

Définition de la fonction affiche()

```
template<typename T> void point<T>::affiche()
{
  cout << "Coordonnées : " << m_x << " " << m_y << endl;
}
```

Déclaration des instances

```
point<int> myPointWithInteger;
point<double> myPointWithDouble(3.2,4.5);
```

Exercice 8

On veut réaliser une fonction qui permet de retourner le minimum de deux arguments (quelque soit le type des arguments).

Rédiger un patron de la fonction min() répondant à l'objectif de l'exercice.

Ajouter la fonction main() pour tester la fonction rédigée.

Exercice 8

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

template<typename T> T min( T & a, T & b)
{
  return a < b ? a : b;
}

int main()
{
  const int i1 = 2, i2 = 7;
  const float f1 = 3.4, f2 = 5.6;
  const char c1 = 'd', c2 = 'z';

  cout << "min(" << i1 << "," << i2 << ") = " << min<int>(i1,i2) << endl;
  cout << "min(" << f1 << "," << f2 << ") = " << min<float>(f1,f2) << endl;
  cout << "min(" << c1 << "," << c2 << ") = " << min<char>(c1,c2) << endl;

  return 0;
}
```

Patron de la fonction min()

```
min(2,7) = 2
min(3.4,5.6) = 3.4
min(d,z) = d
```

Appel de la fonction min()

Exercice 9

Écrire et tester un programme qui instancie un patron de fonction implémentant une recherche dichotomique sur un tableau triés. La fonction reçoit le tableau **v**, l'élément recherché **elem** et les bornes de recherche **inf** et **sup** (inférieure, supérieure). Si trouvé on retourne **son indice** sinon on retourne **–1**.

Exercice 10

On considère une classe **Exemple** contenant une seule méthode **affiche**(tableau,taille).

On veut afficher les éléments d'un tableau de n'importe quel type (int, float, char,…) en connaissant sa taille.

Rédiger un patron de la classe Exemple répondant à l'objectif de l'exercice.

Exercice 10

```cpp
#include<iostream>
using namespace std;
//Déclaration de la classe
template <typename T> class Exemple{
    public:
        void affiche(T *tab, unsigned int nbre);};
//Définition de la méthode de la classe
template <typename T> void Exemple<T>::affiche(T *tab, unsigned int nbre) {
  for(int i = 0; i < nbre; i++)
    cout << tab[i] << " ";
  cout << endl;
}
int main() {
    Exemple<int> A;
  int tabi[6] = {25, 4, 52, 18, 6, 55};
  A.affiche(tabi, 6);
  Exemple<double> B;
  double tabd[3] = {12.3, 23.4, 34.5};
  B.affiche(tabd, 3);
  Exemple<char*> C;
  char *tabs[] = {"Renault", "Peugeot","Ford","Opel"};
  C.affiche(tabs, 4);
  return 0;
}
```

Patron de la classe Exemple

Définition de la méthode affiche()

Création des objets et appel de la fonction affiche()