Académie Régionale d'Education et de Formation Région Souss Massa Draa
Avenue My Youssef Cité NAHDA , BP 7/S Agadir
Web : arefsmd.ac.ma Mail : aref.souss@men.gov.ma

Ministère de l'Education Nationale,
de l'enseignement supérieur, de la formation
des cadres et de la recherche scientifique
وزارة التربية الوطنية
والتعليم العالي وتكوين الأطر
والبحث العلمي
Département de l'Education Nationale
قطاع التربية الوطنية

جامعة القاضي عياض
Faculté des Sciences et Techniques
Marrakech
FST
Marrakech

Programmation Orientée Objet C++

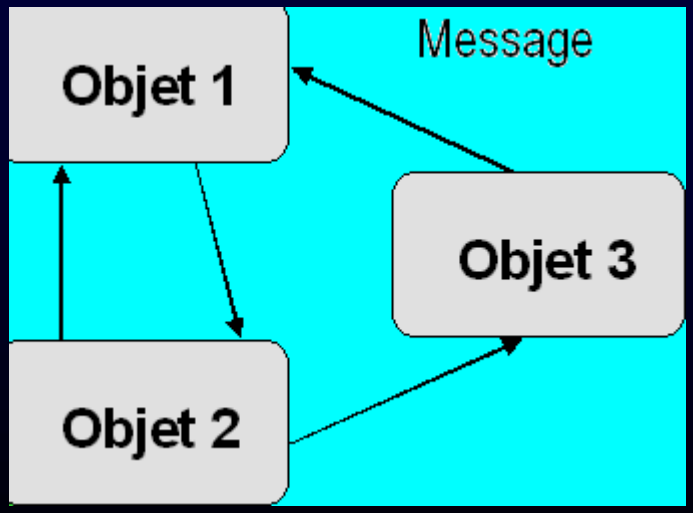
Abdelmounaïm ABDALI

Objectif du cours

- Comprendre les concepts élémentaires de programmation orientée objet (POO)
- Etre capable de lire et comprendre du code C++
- Mettre en oeuvre les concepts de POO en langage C++

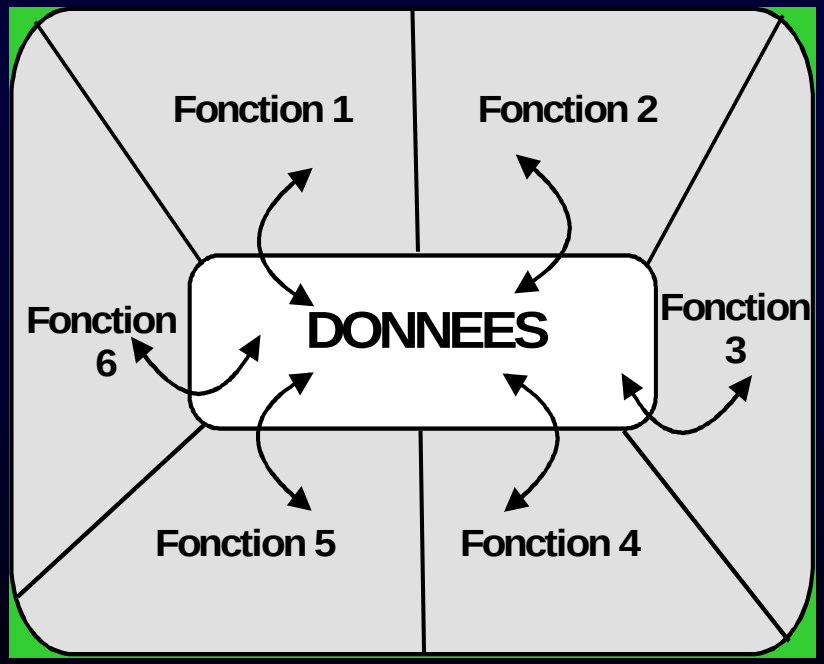
Programmation Orientée Objet (POO)

- **Qu'est ce qu'un Programme Orientée Objet ?**
  - Ensemble d'objets autonomes et responsables qui s'entraident pour résoudre un problème final en s'envoyant des messages.



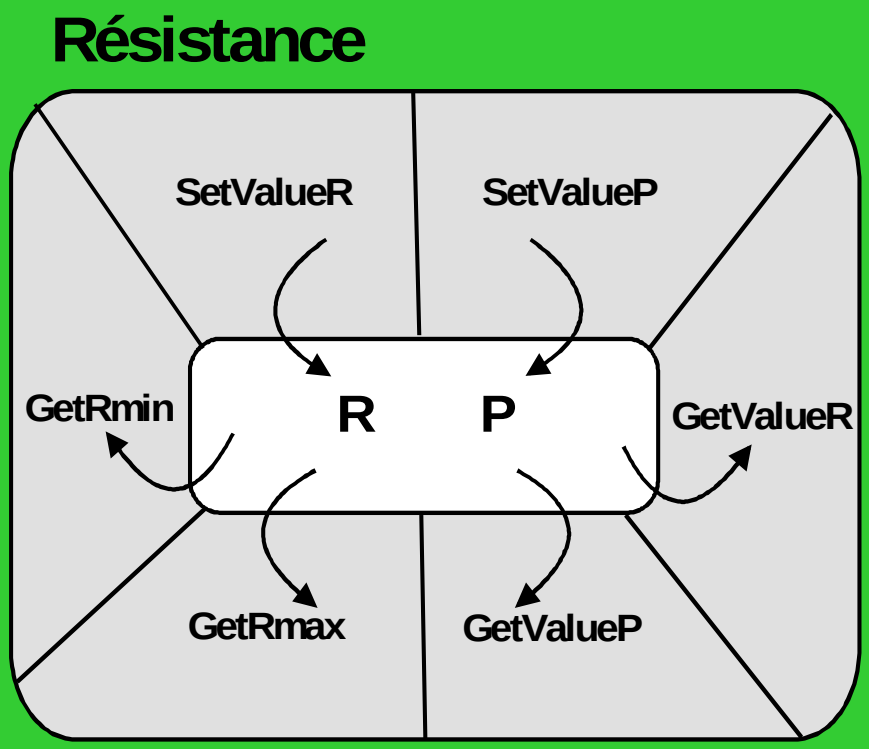
Programmation **O**rientée **O**bjet

- **Qu’est ce qu’un objet**
  - Objet = Données + Méthodes (Fonctions Membres)



Programmation Orientée Objet

- **Exemple d'un objet**



Programmation **O**rientée **O**bjet

- **Encapsulation des données**
  - L'accès aux données des objets est réglementé
    - **Données privées** → accès uniquement par les fonctions membres
    - **Données publiques** → accès direct par l'instance de l'objet
    - **Données protégées** → Mêmes restrictions que s'elles étaient privées, mais elles seront en revanche accessible par les classes filles
  - **Conséquences**
    - Un objet n'est vu que par ses spécifications
    - Une modification interne est sans effet pour le fonctionnement général du programme
    - Meilleure réutilisation de l'objet
  - Exemple de la résistance
    - Les données R et P ne peuvent être utilisées que par les fonctions membres

Programmation **O**rientée **O**bjet

- **Polymorphisme: du Grecque→ plusieurs formes**
  - Un nom (**de fonction, d'opérateur**) peut être associé à plusieurs mais différentes utilisations
  - Exemple
    - Si **a** est un nombre complexe, sqrt(**a**) appellera si elle existe la fonction adaptée au type de **a**.
    - Dans un langage non OO, il aurait fallu connaître le nom de deux fonctions distinctes (selon que **a** soit complexe ou réel)
    - C'est le système qui choisit selon le type de l'argument ou de l'opérande

Programmation **O**rientée **O**bjet

- **L'Héritage**
  - Permet de définir les bases d'un nouvel objet à partir d'un objet existant
  - Le nouvel objet hérite des propriétés de l'ancêtre et peut recevoir de nouvelles fonctionnalités
- **Avantages**
  - Meilleures réutilisations des réalisations antérieures parfaitement au point

Programmation **O**rientée **O**bjet

- **Concept de Classe**
  - **Type réunissant** une description
    - D'une collection de **données membres** hétérogènes ayant un lien entre elles
    - D'une collection de **méthodes (fonctions membres)** servant à manipuler les données membres
  - **Généralisation du type** *structure* ou *record* des langages non **OO** (**classe = définition de données + définition de méthodes**)

Programmation **O**rientée **O**bjet

- **Instance d’une classe**
  - C’est un objet initialisé à partir de la description figée d’une classe
- **Fonctions membres publiques**
  - Les constructeurs
  - Les destructeurs
  - Les accesseurs
  - Les modificateurs

Programmation **O**rientée **O**bjet

Les Constructeurs

1. Permettent d'initialiser l'objet lors de sa création
   - copie des arguments vers les données membres
   - initialisation de variables dynamiques à la création de l'objet
2. Sont appelés de manière automatique à la création de l'objet
3. Peuvent être surchargés→ On peut en définir plusieurs de même nom mais acceptant des arguments différents
4. Possèdent le **même nom** que la classe

Programmation **O**rientée **O**bjet

- **Le Destructeur**
  1. Il est unique
  2. Il est appelé automatiquement lors de la destruction de l'objet
  3. Il sert généralement à éliminer les variables dynamiques de l'objet (créées en général par le constructeur)
  4. Il a pour nom le nom de la classe précédé du symbole **~**

Programmation **O**rientée **O**bjet

- **Les Accesseurs**
  - Catégorie de fonctions membres qui permettent l'accès et l'utilisation des informations privées contenues dans l'objet
- **Les Modificateurs**
  - Permettent de modifier l'état des données internes (publiques ou privées) de l'objet

Programmation Orientée Objet

- **Définition d'une Classe en C++**

```
class nom_classe
{
    private:
      Déclaration des données privées et des
    fonctions membres privées
    public:
      Déclaration des données publiques et des
    fonctions membres publiques
};
```

Programmation Orientée Objet

- **Exemple**

```
class Bouteille
{
    private:
        char Nom [30];
        char couleur[30];
        float contenance;
        float Titre;
        int NBBPES;
    public:
        Bouteille (char *, char *, float,float,int);
        Bouteille (char *,float);
        float GetContenance();
        int GetNBBPES();
};
```

Programmation Orientée Objet

- **Ecriture des fonctions membres en C++**

  Type de l'argument nom de la classe :: nom de la fonction (arguments)

- **Exemple**

```
float Bouteille::GetContenance()
{
        return contenance;
}

Bouteille::Bouteille(char *name, char *color, float c,float t,int n)
{
        strcpy(Nom,name);
        strcpy(couleur,color);
        contenance = c;
        Titre = t;
        NBBPES = n;
}
```

Programmation Orientée Objet

- **Création d'un objet statique en C++**

  nom de la classe nom de l'objet(arguments);

  – Exemple:

  ```
  Bouteille MaBouteille("1890", " noire",33.0,6.5,20);
  ```

- **Création d'un objet dynamique en C++**

  pointeur vers classe = new nom classe(arguments)

  – Exemple:

  ```
  Bouteille* Mabouteille;
  Mabouteille = new Bouteille(" Coca", " noir",25.0,5.5,30);
  delete Mabouteille;
  ```

Programmation **O**rientée **O**bjet

- **Accès aux fonctions membres d'un objet en C++**
  - Si l'objet est représenté par une variable
    - Nom_Variable.Nom fonction();
    - Exemple:
      - Mabouteille.GetContenance();
  - Si l'objet est représenté par un pointeur
    - Nom_pointeur→NomFonction();
    - Exemple:
      - pMabouteille→ GetContenance();

```
#include <iostream.h>
#include <string.h>
class Bouteille
{
      private:
            char NomB[30];
            char couleur[30];
            float contenance;
            float Titre;
            int NBBPES;
      public:
            Bouteille(char *, char *, float,float,int);
            Bouteille(char *,float);
            float GetContenance();
            int GetNBBPES();
            ~Bouteille();
};
float Bouteille::GetContenance()
{
      return contenance;
}

int Bouteille::GetNBBPES()
{
      return NBBPES;
}
```

```
Bouteille::Bouteille(char *name, char *color, float c,float
t,int n)
{
            strcpy(NomB,name);
            strcpy(couleur,color);
            contenance = c;
            Titre = t;
            NBBPES = n;
}

Bouteille::Bouteille(char *name,float c)
{
      strcpy(NomB,name); strcpy(couleur, " Noir");
            contenance = 33.0; TitreA= c;
            NBBPES = 20;
}

Bouteille::~Bouteille()
{
            cout <<"c'est fini !!"<<'\n';
}

main()
{
            Bouteille* Mabouteille;
            Mabouteille = new Bouteille("K",
            " Orange",25.0,5.5,30);
            cout << Mabouteille→GetContenance();
            delete Mabouteille;
}
```

Programmation Orientée Objet

Programmation **O**rientée **O**bjet

- **Héritage en C++**
  - Spécifications Classe Dérivée = Spécifications Classe Ancêtre + Nouvelles Spécifications
  - Héritage des données et des fonctions membres sauf:
    - Les constructeurs et destructeurs de l'ancêtre
  - Introduction d'un nouveau statut: ***protected***
    - membre protégé ne pouvant être utilisé que par les fonctions membres de la classe et de celles qui en seront dérivées

Programmation **O**rientée **O**bjet

- **Syntaxe d'héritage**

class nom_nouvelle_classe : type héritage nom_ancêtre {…};

- **Exemple**
  - On désire créer une classe *BouteilleCoca* à partir de *Bouteille*

class BouteilleCoca : public Bouteille {…}

Programmation Orientée Objet

- **Type d'héritage**
  - Règles de dérivation de classe

mode de dérivation	statut du membre dans la classe ancêtre	statut du membre dans la classe dérivée
private	*private* *protected* *public*	*inaccessible* *private* *private*
protected	*private* *protected* *public*	*inaccessible* *protected* *protected*
public	*private* *protected* *public*	*inaccessible* *protected* *public*

Programmation **O**rientée **O**bjet

- **Conséquences des différents type de dérivation**
  - **Dérivation privée:** (mode par défaut**)**
    - Les membres deviennent *private* ou inaccessible
    - Tous les membres seront inaccessibles à la seconde dérivation
  - **Dérivation publique:**
    - Les membres gardent le même statut sauf les membres *private* qui deviennent inaccessibles
  - **Dérivation protégée:**
    - Tous les membres non privés deviennent de type *protected*, les membres privés deviennent inaccessibles

Programmation **O**rientée **O**bjet

- **Ajout de données et de fonctions membres**

```
class BouteilleCoca : public Bouteille
{
 public:
   BouteilleCoca(char *n, char *coul, float c, float t, int n, int a) :Bouteille(n, coul, c, t, n);

    BouteilleCoca(char * nom, float t) : Bouteille(nom,t) ;
    int GetAnnee();
private:
        int annee;
};
```

Programmation **O**rientée **O**bjet

- **Constructeurs et Destructeur dans les opérations d'héritages**
  - Quand à la fois la classe de base et la classe dérivée ont des constructeurs et des destructeurs:
    - Constructeurs sont exécutés dans l'ordre de la dérivation
    - Destructeurs dans l'ordre inverse de la dérivation
  - C'est au constructeur de l'objet dérivé en dernier de passer les arguments dont a besoin son ancêtre. (voir exemple précédent)

```
#include <iostream.h>
class base {
   int x;
public:
   void setx(int n) {x=n;}
   void showx() {cout <<x<<'\n';}
};
class derive : public base {
   int y;
public:
   void sety(int n) {y=n;}
   void showy() {cout << y << '\n';}
   void show_somme() {cout << x+y <<'\n';}
};
main()
{
   derive ob;
   ob.setx(10);
   ob.sety(20);
   ob.showx();
   ob.showy();
   return(0);
}
```

```
#include <iostream.h>
class base {
   int x;
public:
   void setx(int n) {x=n;}
   void showx() {cout <<x<<'\n';}
};
class derive : private base {
   int y;
public:
   void setxy(int n,int m) {setx(n) ; y=m;}
   void showxy() {showx(); cout << y << '\n';}
};
main()
{
   derive ob;
   ob.setxy(10,20);
   ob.showxy();
   return(0);
}
```

- **Quelques exemples d'héritages**

Programmation Orientée Objet

- **Quelques exemples d'héritages**

```
#include <iostream.h>
class base {
public:
    base(){ cout <<"construction de la classe de base \n;}
   ~base(){ cout <<"destruction de la classe de base \n;}
};
class derive : public base{
public:
    derive() { cout << "construction de la classe dérivée \n";}
    ~derive(){cout << "destruction de la classe dérivée \n";}
};
main()
{
    derive o;
    return(0);
}
```

Affichage Ecran

```
construction de la classe de base
construction de la classe dérivée
destruction de la classe dérivée
destruction de la classe de base
```

• Quelques exemples d'héritages

```
#include <iostream.h>
class base {
public:
    base(){ cout <<"construction de la classe de base \n;}
   ~base(){ cout <<"destruction de la classe de base \n;}
};
class derive : public base{
int j;
public:
   derive(int n) { j = n; cout << "construction de la classe dérivée \n";
}
    ~derive(){cout << "destruction de la classe dérivée \n";}
    showj(){ cout <<j <<'\n';}
};
```

```
main()
{
    derive o(10);
    o.showj();
    return(0);
}
```

Affichage Ecran

```
construction de la classe de base construction de la classe dérivée
 destruction de la classe dérivée destruction de la classe de base
```

Programmation **O**rientée **O**bjet

- **Surcharge des Fonctions**
  - On peut définir des fonctions de même nom mais se différentiant par le nombre et le type d'arguments
  - **Exemple**:
    - float Val_Absolue(float v) {return fabs(v);}
    - int Val_Absolue(int v){return abs(v);}
    - Le système choisira la fonction à appeler selon les arguments.

Programmation **O**rientée **O**bjet

- **Surcharge des opérateurs en C++**
  - Même principe que la surcharge des fonctions
    - Le fonctionnement de l'opérateur s'effectue en fonction du nombre et du type des opérandes
    - La surcharge d'un opérateur doit obligatoirement s'associer à une classe

- **Surcharge des opérateurs en C++**
  - **Exemple:**

```
#include <iostream.h>
class point {
public:
    int x,y;
    point(){x=0;y=0;}
    point(int i,int j) {x=i;y=j;}
    point operator+(point
p2);
};
point
point::operator+(point p2)
{
    point temp;
    temp.x = x + p2.x;
    temp.y = y + p2.y;
    return temp;
}
```

```
main()
{
    point p1(10,10),p2(5,3),p3;
    p3 = p1 + p2;
    cout << p3.x <<< p3.y <<'\n';
}
```

Langage C++

Historique de C++

Créé par B. Stroustrup (Bell Labs.) à partir de 1979 (“C with classes”).
Devient public en 1985 sous le nom de C++.
La version normalisée (ANSI) paraît en 1996.

C++ = C +

- Vérifications de type + stricte
- Surcharge de fonctions
- Opérateurs
- Références
- Gestion mémoire + facile
- Entrées/sorties + facile
- Classes et héritage
- Programmation générique
- …

- **Qu’est-ce que le C++ ?**

  D’après Bjarne Stroustrup, conception du langage C++ pour :

  - Être meilleur en C,
  - Permettre la programmation orientée-objet

- Compatibilité C/C++
  - *C*++ = sur-ensemble de C,
  - C++ ▫ ajout en particulier de l’orienté-objet (classes, héritage, polymorphisme),
  - Cohabitation possible du procédural et de l‘orienté-objet en C++

- Différences C++/Java
  - C++ : langage compilé / Java : langage interprété par la JVM
  - C++ : pas de machine virtuelle et pas de classe de base / *java.lang.object*

Premiers pas en C++

- Programme C++ généralement constitué de plusieurs modules, compilés séparément
- Fichier entête – d’extension .h
  - Contenant les déclarations de types, fonctions, variables et constantes, etc.
  - Inclus via la commande #include
  - Fichier source – d’extension .cpp

Commentaires

C	`/* This is a multi-line` `C comment */`

C++	`/* This is just`
	`the same in C++, but… */`
	`// … We also have single-line comments`

Variable de type référence

- **Possibilité de définir une variable de type *référence***

```
int i = 5;
int & j = i; // j reçoit i
   // i et j désignent le même emplacement mémoire
```

- **Déréférencement automatique** : Application automatique de l'opérateur d'indirection * à chaque utilisation de la référence

```
int i = 5;
int & j = i;    // j reçoit i
int k = j;      // k reçoit 5
j += 2;  // i reçoit i+2 (=7)
j = k;   // i reçoit k (=5)
```

- Initialisation : 2 notations possibles

```
int i = 5;
int i(5);
```

Opérateur new et delete

- 2 opérateurs supplémentaires : **new** et **delete**

```
float *PointeurSurReel = new float;
// Équivalent en C :
//   PointeurSurReel = (float *) malloc(sizeof(float));
int *PointeurSurEntier = new int[20];
// Équivalent en C :
// PointeurSurEntier = (int *) malloc(20 * sizeof(int));
delete PointeurSurReel; // Équivalent en C : free(pf);
delete [] PointeurSurEntier; // Équivalent en C :  free(pi);
```

- **new type** : définition et allocation d'un pointeur de type *type**
- **new type [n]** : définition d'un pointeur de type *type** sur un tableau de *n* éléments de type *type*

passage des paramètres d'une fonction:

Passage des paramètres par valeur

```
#include <iostream>
void echange(int,int);
int main()
{
   int n=10, p=20;
   cout << "avant appel: " << n << " " << p << endl;
   echange(n,p);
   cout << "apres appel: " << n << " " << p << endl;
}
void echange(int a, int b)
{
   int c;
   cout << "debut echange : " << a << " " << b << endl;
   c=a; a=b; b=c;
   cout << "fin echange : " << a << " " << b << endl;
}
```

```
avant appel: 10 20        fin echange: 20 10
debut echange: 10 20      apres appel: 10 20
```

Passage des paramètres par pointeur

```
#include <iostream>

void echange(int*,int*);

int main()
{
int n=10, p=20;
   cout << "avant appel: " << n << " " << p << endl;
   echange(&n,&p);
   cout << "apres appel: " << n << " " << p << endl;
}

void echange(int* a, int* b)
{
   int c;
   cout << "debut echange : " << *a << " " << *b << endl;
   c=*a; *a=*b; *b=c;
   cout << "fin echange : " << *a << " " << *b << endl;
}
```

```
avant appel: 10 20      fin echange: 20 10
debut echange: 10 20    apres appel: 20 10
```

Passage des paramètres par référence

```
#include <iostream>

void echange(int&,int&);

int main()
{

   int n=10, p=20;
   cout << "avant appel: " << n << " " << p << endl;
   echange(n,p);   // attention, ici pas de &n et &p
   cout << "apres appel: " << n << " " << p << endl;
}

void echange(int& a, int& b)
{
   int c;
   cout << "debut echange : " << a << " " << b << endl;
   c=a; a=b; b=c;
   cout << "fin echange : " << a << " " << b << endl;
}
```

```
avant appel: 10 20          fin echange: 20 10
debut echange: 10 20        apres appel: 20 10
```

Fonctions inlines

```
// preprocessor macro
#define MAX(x,y) (x > y) ? x : y;
```

header.h

```
// declaration
inline bool max( int x, int y );


// implementation
inline int max( int x, int y ) {
     return (x > y) ? x : y;
}
```

Déclaration de variables

C++ permet de déclarer des variables n'importe où et de les initialiser.

Règles:

- toujours déclarer ses variables au dernier moment.
- initialiser TOUTES les variables.
- déclarer une variable par ligne.

Entrées / Sorties (I)

C	C++

`#include <stdio.h>`	`#include <iostream>`
	`#include <fstream>`
	`#include <strstream>`

`printf --> standard output`
 	
`scanf <-- standard input`
`fprintf --> FILE*`
`fscanf <-- FILE*`
`sprintf --> char[N]`
`sscanf <-- char*[N]`
	`cout --> standard output`
	`cerr --> standard error output`
	`cin <-- standard input`
	`ofstream --> output file`
	`ifstream <-- input file`
	`ostringstream --> char*[N]`
	`istringstream <-- char*[N]`
	
	`ostream& operator<<`
	`istream& operator>>`

Entrées / Sorties (II)

C

```
#include <stdio.h>
int value = 10;
printf( “value = %d\n”, value );
printf( “New value = ??\n” );
scanf( “%d”, &value );
```

C++

```
#include <iostream>
using namespace std;
int value = 10;
cout << “Value = “ << value << endl;
cout << “New value = ?? ” << endl;
cin >> value;
```

Les opérateurs ‘<<‘ et ‘>>’ sont surchargeables.

Entrées / Sorties (III)

C

```
#include <stdio.h>
FILE* file;
if( fopen( file, “filename” ) ) {
    fprintf( file, “Hello!!\n” );
}
```

C++

```
#include <fstream>
using namespace std;
ofstream outfile( “filename” );
if( outfile.good() ) {
    outfile << “Hello!!” << endl;
}
```

Les Classes

- **Classe :**
  - Regroupement de données (attributs ou champs) et de méthodes (fonctions membres)
  - Extension des structures (`struct`) avec différents niveaux de visibilité (`protected`, `private` et `public`)
- En programmation orientée-objet pure: encapsulation des données et accès unique des données à travers les méthodes
- **Objet** : instance de classe
  - Attributs et méthodes communs à tous les objets d'une classe
  - Valeurs des attributs propres à chaque objet
- **Encapsulation**
  - Caractérisation d'un objet par les spécifications de ses méthodes : interface
  - Indépendance vis à vis de l'implémentation

1er exemple de classe

```
class Horaire{

  private : // déclaration des membres privés
            // private: est optionnel (privé par défaut)
     int heure;      // de 0 à 24
     int minute;     // de 0 à 59
     int seconde;    // de 0 à 59

  public : // déclaration des membres publics
     Horaire(); // Constructeur
     void SetHoraire(int, int, int);
     void AfficherMode12h();
     void AfficherMode24h();
};
```

Interface de la classe

Constructeur

Constructeur : Méthode appelée automatique à la création d'un objet

```
Horaire::Horaire() {heure = minute = seconde = 0;}
```

Définition d'un constructeur ⇒ Création d'un objet en passant le nombre de paramètres requis par le constructeur

```
int main()
{ Horaire h;  // Appel du constructeur qui n'a pas de paramètre
  …
}
```

Si on avait indiqué dans la définition de la classe :

```
Horaire (int = 0, int = 0, int = 0);
```

- **Définition du constructeur :**

```
Horaire:: Horaire (int h, int m, int s)
 { SetHoraire(h,m,s);}
```

- **Déclaration des objets :**

```
Horaire h1, h2(8), h3 (8,30), h4 (8,30,45);
```

Constructeur et destructeur

```
class Exemple
{
    public :
      int attribut;
      Exemple(int); // Déclaration du constructeur
      ~Exemple();   // Déclaration du destructeur
} ;

Exemple::Exemple (int i)  // Définition du constructeur
{ attribut = i;
  cout  <<  "**  Appel  du  constructeur  -  valeur  de
   l'attribut = " << attribut << "\n";
}

Exemple::~Exemple()  // Définition du destructeur
{ cout << "** Appel du destructeur - valeur de l'attribut
   = " << attribut << "\n";
}
```

Constructeur par recopie (*copy constructor*) :

- Constructeur créé par défaut mais pouvant être redéfini
- Appelé lors de l'**initialisation d'un objet par recopie** d'un autre objet, lors du **passage par valeur d'un objet** en argument de fonction ou en **retour d'un objet** comme retour de fonction

```
MaClasse c1;
MaClasse c2=c1; // Appel du constructeur par recopie
```

- Possibilité de définir explicitement un constructeur par copie si nécessaire :
  - Un seul argument de type de la classe
  - Transmission de l'argument par référence

```
MaClasse(MaClasse &);
MaClasse(const MaClasse &) ;
```

```
class Exemple
{
    public :
      int attribut;
      Exemple(int); // Déclaration du constructeur
      ~Exemple();   // Déclaration du destructeur
} ;

int main()
{  Exemple e(1); // Déclaration d'un objet Exemple
   Exemple e2=e; // Initialisation d'un objet par recopie
   return 0;
}
```

```
// Reprise de la classe Exemple
class Exemple
{ public :
        int attribut;
        Exemple(int);
      // Déclaration du constructeur par recopie
      Exemple(const Exemple &)
      ~Exemple();
} ;

// Définition du constructeur par recopie
Exemple::Exemple (const Exemple & e)
{ cout << "** Appel du constructeur par recopie ";
  attribut = e.attribut;        // Recopie champ à champ
  cout << " - valeur de l'attribut après recopie = " << attribut <<
endl;
}
```

Résultat de l'exécution du programme avant la définition explicite du constructeur par recopie :

```
** Appel du constructeur - valeur de l'attribut = 1
** Appel du constructeur par recopie  - valeur de l'attribut après recopie= 1
** Appel du destructeur - valeur de l'attribut = 1
** Appel du destructeur - valeur de l'attribut = 1
```

Surcharge des méthodes

- **Surcharge des méthodes**

```
MaClasse();                 Afficher();
MaClasse(int);              Afficher(char* message);
```

- Possibilité de définir des **arguments par défaut**

```
MaClasse(int = 0);          Afficher(char* = "" );
```

- Possibilité de définir des **méthodes en ligne**

```
inline MaClasse::MaClasse() {corps court};

class MaClasse
{ …
 MaClasse() {corps court};
}
```

Définition de la méthode dans la déclaration même de la classe

Incorporation des instructions correspondantes (en langage machine) dans le programme ⇒ plus de gestion d'appel

Passage des paramètres objets

- **Transmission par valeur**

```
bool Horaire::Egal(Horaire h)
  { return ((heure == h.heure) && (minute == h.minute)
   && (seconde == h.seconde)) ;
  }

int main()
{ Horaire h1, h2;
  …
   if (h1.Egal(h2)==true)
        // h2 est recopié dans un emplacement
        // local à Egal nommé h
        // Appel du constructeur par recopie
}
```

- **Transmission par référence**

```
bool Egal(const Horaire & h)
```

Méthode retournant un objet

- **Transmission par valeur**

```
Horaire Horaire::HeureSuivante()
 { Horaire h;
    if (heure<24) h.SetHeure(heure+1);
       else h.SetHeure(0);
    h.SetMinute(minute); h.SetSeconde(seconde);
    return h; // Appel du constructeur par recopie
 }
```

- **Transmission par référence**

```
Horaire & Horaire::HeureSuivante()
```

La variable locale a la méthode est détruite à la sortie de la méthode – Appel automatique du destructeur!

Propriétés des méthodes

- **Auto-référence : pointeur `this`**
  - Pointeur sur l'objet (i.e. l'adresse de l'objet) ayant appelé
  - Uniquement utilisable au sein des méthodes de la classe

```
Horaire::AfficheAdresse()
{ cout << "Adresse : " << this ;
}
```

- **Qualificatif statique : `static`**
  - Applicable aux attributs et aux méthodes
  - Définition de propriété indépendante de tout objet de la classe

Exemple

```
class Horaire{
  private :
    static int nbObjets; // compteur (statique)
    int heure;
    int minute;
    int seconde;
  public :
    Horaire();
    static void AfficherNbObjets()
     {cout << nbObjets ;}
    void SetHoraire(int, int, int);
    void AfficherMode12h();
    void AfficherMode24h();
};
```

Méthode constante

Méthode constante

- Utilisable pour un objet déclaré constant
- Pour les méthodes ne modifiant pas la valeur des objets

```
class Horaire
{ …
  void Afficher() const ;
  void AfficherMode12h();
}
```

Fonction main

```
const Horaire h;
h.Afficher(); // OK
h.AfficherMode12h() // KO
          // h est const mais pas la méthode!!
Horaire h2;
h2.Afficher(); //OK
```

Surcharge d'opérateurs

Notions de **méthode amie** : **`friend`**

- Fonction extérieure à la classe ayant accès aux données privées de la classes
- Contraire à la P.O.O. mais utile dans certain cas
- Plusieurs situations d'« amitié » [Delannoy, 2001] :
  - Une fonction indépendante, amie d'une classe
  - Une méthode d'une classe, amie d'une autre classe
  - Une fonction amie de plusieurs classes
  - Toutes les méthodes d'une classe amies d'une autre classe

```
friend type_retour NomFonction (arguments) ;
// A déclarer dans la classe amie
```

Fonctions amies indépendantes

- Une fonction est *l'amie* (*friend*) d'une classe lorsqu'elle est autorisée à adresser directement les membres privés de cette classe. Pour la déclarer ainsi, il faut donner, à l'intérieur de la classe, la déclaration complète de la fonction précédée du mot clé friend.
- Voici un exemple simple :

```
class exemple {
      int i, j;
      public:
      exemple() { i = 0; j = 0; }
      friend exemple inverse(exemple);
      };

exemple inverse(exemple ex)
//  renvoie ex avec tous les bits inversés
{
      exemple ex2 = ex;
      ex2.i = ~ex2.i;          //  accès aux champs
      ex2.j = ~ex2.j;
      return ex;
}
```

Fonctions membre d'une classe amie à une autre classe

- On souhaite parfois qu'une méthode d'une classe puisse accéder aux parties privées d'une autre classe. Pour cela, il suffit de déclarer la méthode friend également, en utilisant son nom complet (nom de classe suivi de **::** et du nom de la méthode). Par exemple :

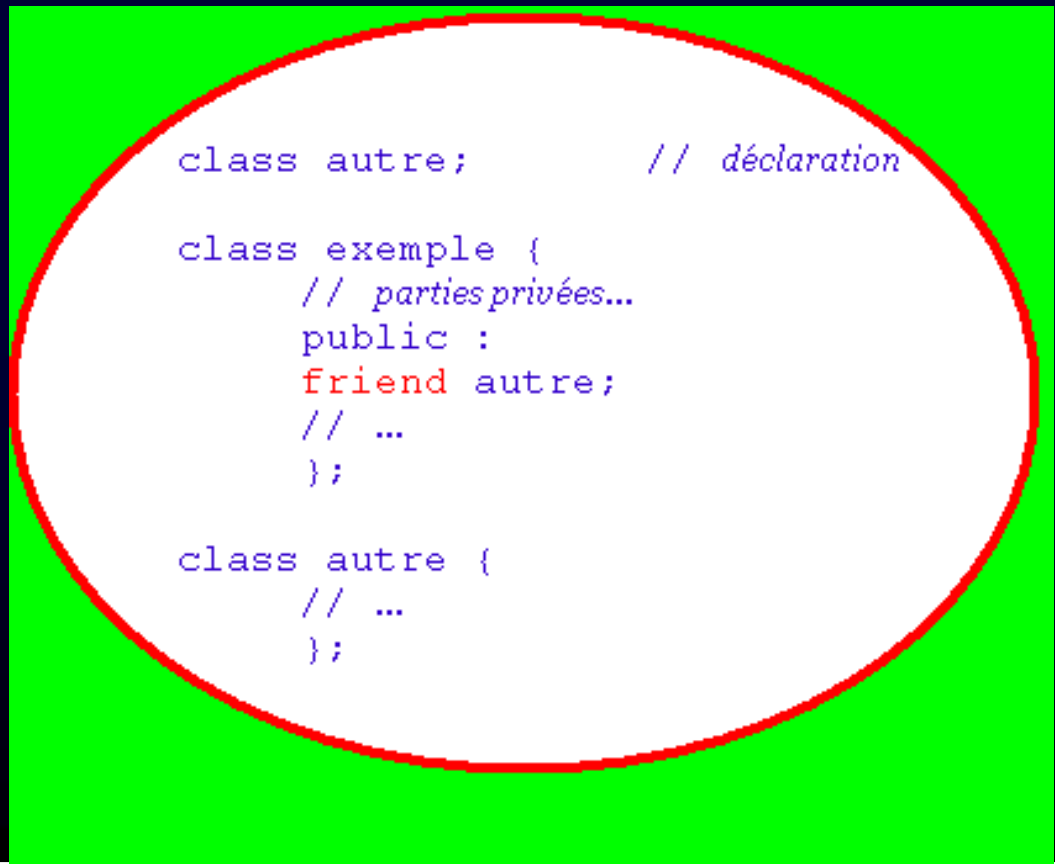
```
class autre {
      // …
      void combine(exemple);
      };

class exemple {
      // …parties privées
      public :
      friend void autre::combine(exemple);
      };

void autre::combine(exemple ex)
{
      // utilise les membres privés de ex
}
```

Classes amies

- Lorsqu'on souhaite que tous les membres d'une classe puissent accéder aux parties privées d'une autre classe, on peut déclarer « amie » une classe entière :



```
class autre;          // déclaration

class exemple {
      // parties privées...
      public :
      friend autre;
      // ...
      };

class autre {
      // ...
      };
```

Surcharge d'opérateurs

Possibilité en C++ de redéfinir n'importe quel opérateur unaire ou binaire : =, ==, +, -, *, \, [], (), <<, >>, ++, -- , +=, -=, *=, /= , & etc.

```
class Horaire
{ …
  bool operator== (const Horaire &);
}

bool Horaire::operator==(const Horaire& h)
{
  return((heure==h.heure) && (minute ==
  h.minute) && (seconde == h.seconde));
}
```

Fonction main

```
Horaire h1, h2;
…
if (h1==h2) …
```

Surcharge d'opérateur

```
class Point
{
 int x;
 int y;
public :
 Point(){}
 Point(int a, int b){ x=a; y=b; }
 Point operator +(const Point & a);
 Point operator -(const Point & a);
 int operator==(const Point & p);

 void Affiche()
 {
  // "this" est un pointeur sur la classe même
  cout << this << "->" << x << ", " << y << endl;
 }
};

Point Point::operator +(const Point & a)
{ // Addition de 2 points
 Point p;
 p.x = x + a.x;
 p.y = y + a.y;

 return p;
}
Point Point::operator -(const Point & a)
{ // Soustraction de 2 points
 Point p;
 p.x = x - a.x;
 p.y = y - a.y;

 return p;
}

int Point::operator==(const Point & p)
{ // Egalité de 2 points (remplace "Coincide")
 if( x==p.x && y==p.y )
  return 1;
 else
  return 0;
}

void main()
{
 Point p(1,2);
 p.Affiche();
 Point pp(3,4);
 pp.Affiche();
 Point ppp = p+pp;
 ppp.Affiche();

 if( p==pp )
  cout << "p==pp" << endl;
 else
  cout << "p!=pp" << endl;

 p = ppp;
 p.Affiche();
 pp = p-ppp;
 pp.Affiche();

 if( p==ppp )
  cout << "p==ppp" << endl;
 else
  cout << "p!=ppp" << endl;
}
```

Objet membre

Possibilité de créer une classe avec un membre de type objet d'une classe

```
// exemple repris de [Delannoy, 2004]
class point
 {
   int abs, ord ;
   public :
    point(int,int);
 };
 class cercle
 {
   point centre; // membre instance de la classe point
   int rayon;
   public :
    cercle (int, int, int);
 };
```

```
#include "ObjetMembre.h"

point::point(int x=0, int y=0)
{
   abs=x; ord=y;
   cout << "Constr. point " << x << " " << y << endl;
}

cercle::cercle(int abs, int ord, int ray) : centre(abs,ord)
{
   rayon=ray;
   cout << "Constr. cercle " << rayon << endl;
}

int main()
{
   point p;
   cercle c (3,5,7);
}
```

Affichage :

```
Constr. point 0 0
Constr. point 3 5
Constr. cercle 7
```

Autre manière d'écrire le constructeur

```
// Autre manière d'écrire le constructeur de la classe cercle
cercle::cercle(int abs, int ord, int ray)
{
    rayon=ray;
    // Attention création d'un objet temporaire point
    centre = point(abs,ord);
    cout << "Constr. cercle " << rayon << endl;
}

int main()
{
    point p = point(3,4); // ⇔ point p (3,4);
    // ici pas de création d'objet temporaire
    cercle c (3,5,7);
}
```

Affichage :

```
Constr. point 3 4
Constr. point 0 0
Constr. point 3 5
Constr. cercle 7
```

L’héritage

- **Héritage** [Delannoy, 2004] :
  - Un des fondements de la P.O.O
  - A la base des possibilités de réutilisation de composants logiciels
  - Autorisant la définition de nouvelles classes « dérivées » à partir d'une classe existante « de base »
- **Super-classe** ou classe mère
- **Sous-classe** ou classe fille : spécialisation de la super-classe - héritage des propriétés de la super-classe
- Possibilité d'héritage multiple en C++

Exemple

```
class CompteBanque
{
    long ident;
    float solde;
    public:
     CompteBanque(long id, float so = 0);
     void deposer(float);
     void retirer(float);
     float getSolde();
};

class ComptePrelevementAuto : public CompteBanque
{
    float prelev;
    public:
     void prelever();
     ComptePrelevementAuto(long id, float pr, float so);
 };
```

```
void transfert(CompteBanque cpt1, ComptePrelevementAuto cpt2)
{
    if (cpt2.getSolde() > 100.00)
    {
        cpt2.retirer(100.00);
        cpt1.deposer(100.00);
    }
}

void ComptePrelevementAuto::prelever()
{
    if (solde > 100.00)
    {
      // KO: solde est un champ privé de la super-classe
    }
}
```

Une sous-classe n'a pas accès aux membres privés de sa super-classe!!

Héritage simple et constructeurs

```
// Exemple repris de [Delannoy, 2004] page 254
#include <iostream>
using namespace std ;

// ************ classe point **********************
class point
{
   int x, y ;
   public :
    // constructeur de point ("inline")
    point (int abs=0, int ord=0)
     { cout << "++ constr. point : " << abs << " " << ord << endl;
       x = abs ; y =ord ;
     }
    ~point () // destructeur de point ("inline")
     { cout << "-- destr.  point :    " << x << " " << y << endl ;
     }
} ;
```

```
// ************ classe pointcol ******************
class pointcol : public point
{
   short couleur ;
   public :
    pointcol (int, int, short) ;  // déclaration constructeur pointcol
   ~pointcol ()                   // destructeur de pointcol ("inline")
     { cout << "-- dest. pointcol - couleur : " << couleur << endl ;
     }
} ;

pointcol::pointcol (int abs=0, int ord=0, short cl=1) : point (abs, ord)
{
    cout << "++ constr. pointcol : " << abs << " " << ord << " " << cl
      << endl ;
    couleur = cl ;
}
```

```
// ************ programme d'essai ****************
int main()
{
    pointcol a(10,15,3) ;                // objets non dynamiques
    pointcol b (2,3) ;
    pointcol c (12) ;
    pointcol * adr ;
    adr = new pointcol (12,25) ;         // objet dynamique
    delete adr ;
}
```

Résultat :

```
++ constr. point :    10 15
++ constr. pointcol : 10 15 3
++ constr. point :    2 3
++ constr. pointcol : 2 3 1
++ constr. point :    12 0
++ constr. pointcol : 12 0 1
++ constr. point :    12 25
++ constr. pointcol : 12 25 1
-- dest. pointcol - couleur : 1
-- destr.  point :    12 25
-- dest. pointcol - couleur : 1
-- destr.  point :    12 0
-- dest. pointcol - couleur : 1
-- destr.  point :    2 3
-- dest. pointcol - couleur : 3
-- destr.  point :    10 15
```

```
// ************ programme d'essai ****************
int main()
{
   pointcol a(10,15,3) ;                // objets non dynamiques
   pointcol b (2,3) ;
   pointcol c (12) ;
   pointcol * adr ;
   adr = new pointcol (12,25) ;         // objet dynamique
   delete adr ;
}
```

Résultat :

```
++ constr. point :    10 15
++ constr. pointcol : 10 15 3
++ constr. point :    2 3
++ constr. pointcol : 2 3 1
++ constr. point :    12 0
++ constr. pointcol : 12 0 1
++ constr. point :    12 25
++ constr. pointcol : 12 25 1
-- dest. pointcol - couleur : 1
-- destr.  point :    12 25
-- dest. pointcol - couleur : 1
-- destr.  point :    12 0
-- dest. pointcol - couleur : 1
-- destr.  point :    2 3
-- dest. pointcol - couleur : 3
-- destr.  point :    10 15
```

Héritage simple et redéfinition/sur-définition

```
class Base
 {
    protected :
    int a;
    char b;
    public :
     …
     void affiche();
 };

class Derivee : public Base
 {
    float a; // redéfinition de l'attribut a
    public :
     …
     void affiche(); // redéfinition de la méthode affiche
     float GetADelaClasseDerivee(){return a;} ;
     int GetADeLaClasseDeBase() {return Base::a;}
 };
```

```
void Base::affiche()
{ cout << a << b << endl; }

void Derivee::affiche()
{ // appel de affiche
  // de la super-classe
  Base::affiche();
  cout << "a est un réel";
}
```

Héritage simple et redéfinition/sur-définition

```
class A
 {   …
     public :
      void f (int);
      void f (char);
      void g (int);
      void g (char);
      …
 };
class B : public A
 {   …
     public :
      void f (int);
      void f (float);
        …
 };
```

```
int main()
 {
     int n;
     float x;
     char c;
     B b;
     …
     b.f(n); // appel de B::f(int)
     b.f(x); // appel de B::f(float)
     b.f(c); // appel de B::f(int)
     …// avec conversion de c en int
     …// pas d'appel à A::f(int)
     …// ni d'appel à A::f(char)
     …
     b.g(n); // appel de A::g(int)
     b.g(x); // appel de A::g(float)
     b.g(c); // appel de A::g(char)

 }
```

Héritage simple et amitié

```
class A
{    friend class ClasseAmie;
     public:
       A(int n=0): attributDeA(n) {}
     private:
       int attributDeA;
 };
class ClasseAmie
{  public:
       ClasseAmie(int n=0): objetMembre(n) {}
       void affiche1() {cout << objetMembre. attributDeA << endl;}
                              // OK: Cette classe est est amie de A
     private:
       A objetMembre;
};
class ClasseDérivée: public ClasseAmie
{      public:
         ClasseDérivée(int x=0,int y=0): ClasseAmie(x), objetMembre2(y) {}
         void Ecrit() { cout << objetMembre2.attributDeA << endl; }
                        // ERREUR: ClasseDérivée n'est pas amie de A
                        // error: `int A::attributDeA' is private
       private:
         A objetMembre2;
```

Compatibilité entre classe de base et classe dérivée

- Possibilité de **convertir implicitement une instance d'une classe dérivée en une instance de la classe de base, si l'héritage est public**

- **L'inverse n'est pas possible : impossibilité** de convertir une instance de **la classe de base en instance de la classe dérivée**

```
ClasseDeBase a;
ClasseDérivée b;
a=b; // légal et appel de l'opérateur = de la classe de Base
     // s'il a été redéfini ou de l'opérateur par défaut sinon
b=a; // illégal
     // error: no match for 'operator=' in 'b = a'
```

Héritage multiple

- Possibilité de créer des classes dérivées à partir de plusieurs classes de base
- Pour chaque classe de base : possibilité de définir le mode d'héritage
- **Appel des constructeurs dans l'ordre de déclaration de l'héritage**
- **Appel des destructeurs dans l'ordre inverse de celui des constructeurs**

```
class Point
{
  int x;
  int y;
  public:
   Point(...){...}
   ~Point(){...}
   void affiche(){...}
};
```

```
class Couleur
{
  int coul;
  public:
    Couleur(...){...}
    ~Couleur(){...}
    void affiche(){...}
};
```

```
// classe dérivée de deux autres classes
class PointCouleur : public Point, public Couleur
{
  ...
   // Constructeur
   PointCouleur (...) : Point(...), Couleur(...)
```

```
int main()
{  PointCouleur p(1,2,3);
   cout << endl;
   p.affiche(); // Appel de affiche() de PointCouleur
   cout << endl;
   // Appel "forcé" de affich() de Point
   p.Point::affiche();
   cout << endl;
   // Appel "forcé" de affiche() de Couleur
   p.Couleur::affiche();
}
```

```
int main()
{   PointCouleur p(1,2,3);
    cout << endl;
    p.affiche(); // Appel de affiche() de PointCouleur
    cout << endl;
    // Appel "forcé" de affich() de Point
    p.Point::affiche();
    cout << endl;
    // Appel "forcé" de affiche() de Couleur
    p.Couleur::affiche();
}
```

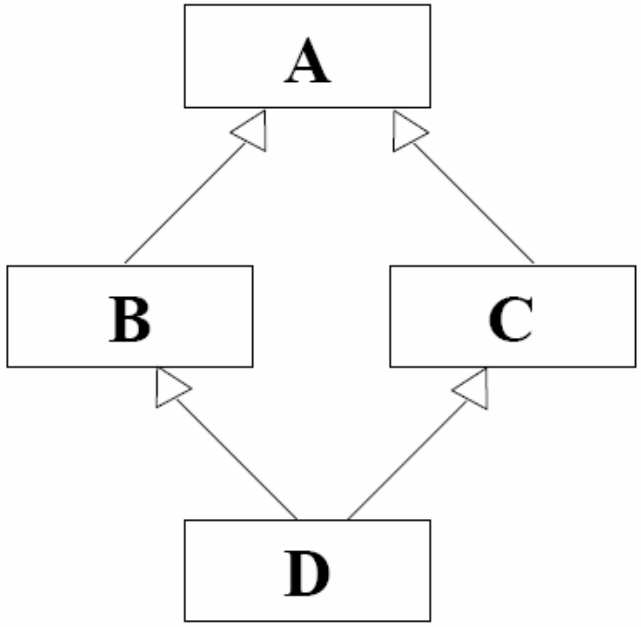
```
** Point::Point(int,int)
** Couleur::Couleur(int)
** PointCouleur::PointCouleur(int,int ,int)
Coordonnées : 1 2
Couleur : 3
Coordonnées : 1 2
Couleur : 3
** PointCouleur::~PointCouleur()
** Couleur::~Couleur()
** Point::~Point()
```

Héritage multiple

```
class A
{ int x, y;
   ....
};

class B : public A {....};
class C : public A {....};

class D : public B, public C
{ ....
};
```



Héritage virtuel: Fonctions virtuelles

Considérons une classe dérivée **Derivee** d'une classe de base **Base** :

```
class Base {
   private :
         int   age ;
   public :
         Base(int k) { age = k ;}
               Base(){}
               int getAge() { return age; }
               void afficher() {
              cout << "afficher dans
Base : " << age << endl << endl;
               }
};
```

```
class Derivee : public Base {
    private :
            double taille ;
    public :
   Derivee(int k, double t) : Base(k) {taille =
t }
      void afficher() {
 cout << "afficher dans Derivee : " <<
getAge()
<< " an(s) " << taille << " metre " << endl <<
endl;
                }
};
```

- Avec les déclarations :
  - Base pers1(23);
  - Derivee pers2(28, 1.72);
    - Base * p1 = &pers1 ;
    - p1->afficher(); appelle la méthode **afficher()** de la classe Base.
- Si on fait (autorisé) :
  - p1=&pers2;
- p1 pointe sur un objet de la classe **Derivee.** L'instruction p1->afficher(); **va faire toujours appel à la méthode afficher de la classe Base**, alors que **Derivee** possède lui aussi une méthode **afficher**.

- **Raison:** Le choix de la méthode à appeler a été réalisé lors de la compilation; il a donc été fait en fonction du type de l'objet *p1*.
- On parle alors de **ligature statique**.
  - Le typage statique est le type par défaut en C++

- Si on veut que *p1 -> afficher()* appelle non plus systématiquement la méthode *afficher* de la classe Base, mais celle correspondant au type de l'objet réellement désigné par *p1* (Derivee) on doit utiliser une **ligature dynamique** :

- Le choix se fera à l'exécution et donc la fonction est sélectionnée en fonction de la classe pointée par le pointeur p1.

- Le mot clé **virtual** force la ligature dynamique :

```
class Base {
        private :
                int   age ;
        public :
                Base(int k) {
                        age = k ;
                }
                Base(){}
                int getAge() { return age; }
                virtual void afficher() {
                     cout << "afficher dans Base : " << age << endl << endl;
                }
};
class Derivee : public Base {
        private :
                double taille ;
        public :
                Derivee(int k, double t) : Base(k) {
                        taille = t ;
                }
                //il n'est pas nécessaire de déclarer dans une classe dérivée
                // le mot virtual devant une méthode déclarée virtual dans la
                // classe de base
                void afficher() {
                     cout << "afficher dans Derivee : " << getAge()
                          << " an(s) " << taille << " metre "
                          << endl << endl;
                }
};
void main() {
       Derivee pers2(28, 1.72);
 Base * p1 = &pers2;
       p1->afficher();
}
/* Exécution :
afficher dans Derivee : 28 an(s) 1.72 metre
*/
```

Classes abstraites

- **Pourquoi ?**
  - Lors de la conception d'une hiérarchie, on a besoin de créer des classes plus générales d'abord et différer l'implémentation à des étapes plus éloignées quand le comportement est très bien défini.

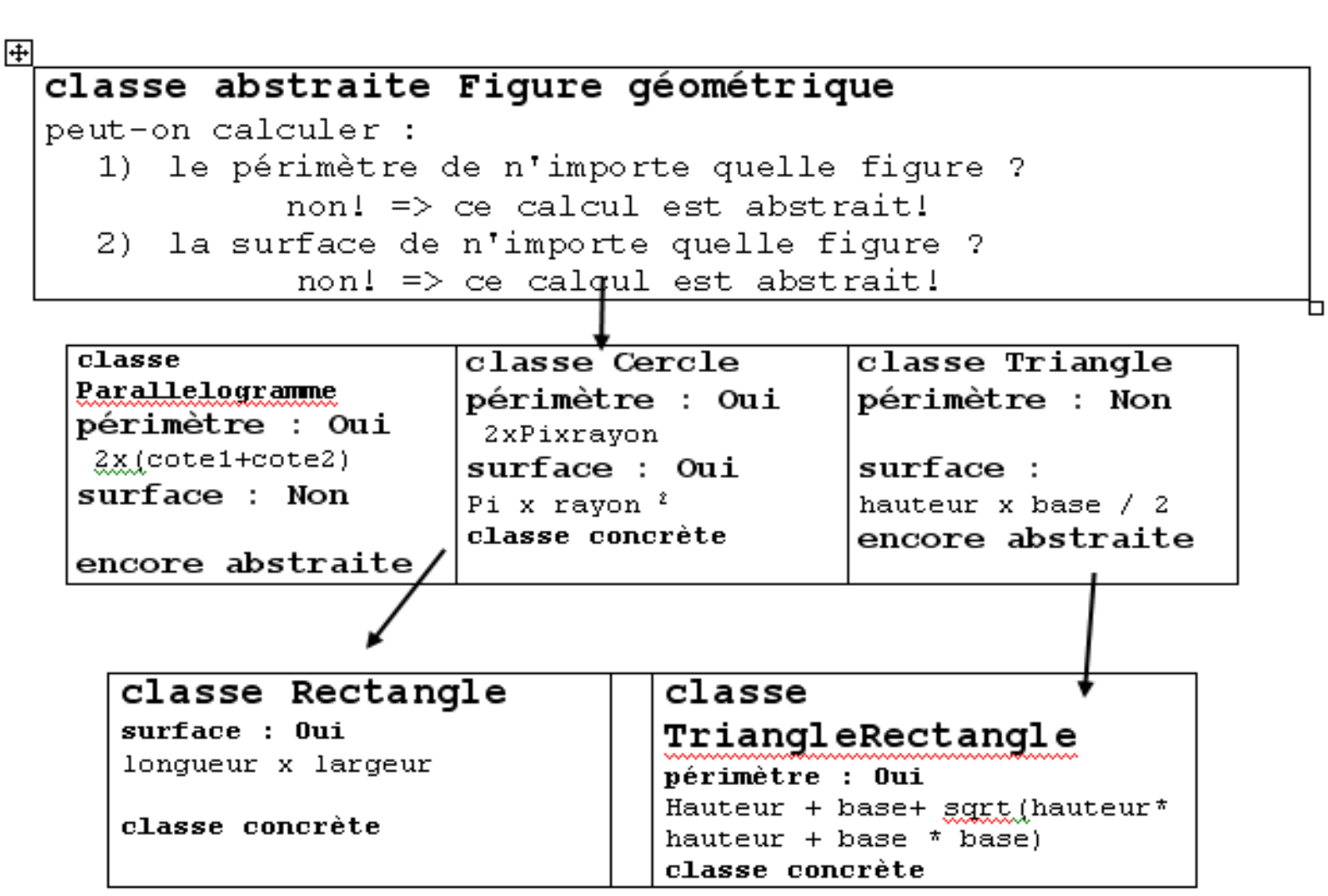
Classes abstraites

- **Définition :** Une classe est dite **abstraite** si elle contient au moins une fonction virtuelle pure (méthode abstraite) :

```
class X {
        // afficher est une fonction virtuelle pure
        virtual void afficher() =0;
};
```

Règles à respecter :

- **Pas d'instanciation : trop abstraite pour la construction d'un objet**
- Une classe abstraite n'existe que pour être héritée.
- **Une fonction virtuelle pure (méthode abstraite)** déclarée dans la classe abstraite doit être implémentée dans une de ses sous-classes (pas nécessairement dans toutes ses sous-classes) :


classe abstraite Figure géométrique
peut-on calculer :
1) le périmètre de n'importe quelle figure ?
non! => ce calcul est abstrait!
2) la surface de n'importe quelle figure ?
non! => ce calcul est abstrait!
classe Parallelogramme
périmètre : Oui
2x(cote1+cote2)
surface : Non
encore abstraite
classe Cercle
périmètre : Oui
2xPixrayon
surface : Oui
Pi x rayon ²
classe concrète
classe Triangle
périmètre : Non
surface :
hauteur x base / 2
encore abstraite
classe Rectangle
surface : Oui
longueur x largeur
classe concrète
classe TriangleRectangle
périmètre : Oui
Hauteur + base+ sqrt(hauteur*
hauteur + base * base)
classe concrète

```
#include <iostream.h>
#include <math.h>
const double PI=3.14;

class FigureGeometrique {
public:
	virtual double perimetre()=0;
	virtual double surface()=0;
	virtual void identifier(char * message) {
		cout << "FigureGeometrique: " << message << endl;
	}
	void afficher(char *message) {
		identifier(message);
		cout << "-perimetre: " << perimetre() << endl;
		cout << "-surface: " << surface() << endl;
	}
};
```

```
class Cercle :public FigureGeometrique {
private:
	double rayon;
public:
	Cercle(double r) {
		rayon=r;
	}
    void identifier(char *message) {
		cout << "Cercle: " << message << endl;
	}
	double perimetre() {
		return 2* PI*rayon;
	}
	double surface() {
		return PI*rayon*rayon;
	}
};
```

```
void main()
{
	Cercle C(8.7);
	C.afficher("C");

}
```

Polymorphisme

- Le mot polymorphisme est issu d’un mot grec signifiant "**qui possède plusieurs formes**"
- En C++, le but de polymorphisme est de permettre d’identifier clairement lequel des champs, lequel des objets, laquelle des méthodes qu’on travaille lors d’une possibilité de confusion dans leur utilisation.

Polymorphisme

- Il y a trois cas de polymorphisme :
  - Autoréférence **this** (this pointe vers l'objet courant)
  - Appel d'une méthode de la classe de base quand la classe dérivée dispose d'une méthode portant le même nom (**redéfinition d'une méthode**).
  - Cas de **fonctions virtuelles**

```
#include <iostream.h>
#include <math.h>
class Triangle
{
    protected :
      double cote1, cote2, cote3 ;
    public :
      Triangle (double cote1, double cote2,
       double cote3) {
 this->cote1 = cote1 ; // cas 1 de polymorphisme
                    this->cote2 = cote2 ;
                    this->cote3 = cote3 ;
          }
      Triangle () {}
      void afficher(char * message) {
          cout << message ;
         cout << "<" << cote1 << ", "
                    << cote2 << ", " << cote3 << ",
       perimetre : "
                    << perimetre() << ">\n\n";
      }
      double perimetre() { return cote1 + cote2 +
       cote3 ; }
 } ;
class TriangleRectangulaire : public Triangle
{
        private :
       double hauteur, base ;
 public :
          TriangleRectangulaire (double h, double
        b) {
                    hauteur = h;
                    base = b;
          }
          TriangleRectangulaire () {}
          double surface() {
                     return hauteur * base / 2.0 ;
          }
     double perimetre() {
                    return hauteur + base + sqrt(hauteur*hauteur
    +
                                                base * base);
          }

          void afficher(char * message) {
                cout << message ;
                cout << "<hauteur :" << hauteur << ", base : "
                     << base << ", perimetre : "
                     << perimetre() << ", surface : " << surface()
                     << ">\n\n";
          }

     } ;


void main()
{  Triangle  t1(5, 10, 20);
   t1.afficher("Informations du triangle regulier t1 :");
   TriangleRectangulaire t2(10, 6);
   t2.afficher("Informations du triangle rectangulaire t2
:\n");

}
/* Exécution :
Informations du triangle regulier t1 :<5, 10, 20,
perimetre : 35>
Informations du triangle rectangulaire t2 :
<hauteur :10, base : 6, perimetre : 27.6619, surface : 30>
```

Polymorphisme:
cas des fonctions virtuelles
(déjà traité)

Patrons de classe

- Un **patron de classe** est un modèle que le compilateur utilise pour créer des classes au besoin.

- Les patrons de classe sont très utiles pour la **réutilisation** des classes conteneurs dont le rôle est de regrouper des objets suivant une certaine structure.

Exemple de la classe Point

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

// classe des points dont les coordonnées sont des entiers
class PointEntier {

  private :
    int x, y ;

  public :
    PointEntier (int a = 0, int b = 0) {
      x = a, y = b ;
    }

    double longueur () const {
      return  sqrt( x * x + y * y);
                                  }

    friend void afficher(const PointEntier &) ;
};

void afficher(const PointEntier & P) {
     cout << "<x = " << P.x << ", y = " << P.y
          << ", longueur = " << P.longueur() << ">\n\n";
}
```

```
// classe des points dont les coordonnées sont des réels
class PointReel {

  private :
    double x, y ;

  public :
    PointReel (double a = 0, double b = 0) {
      x = a, y = b ;
    }

    double longueur () const {
      return  sqrt( x * x + y * y);
                                  }

    friend void afficher(const PointReel &) ;
};

void afficher(const PointReel & P) {
     cout << "<x = " << P.x << ", y = " << P.y
          << ", longueur = " << P.longueur() << ">\n\n";
}

// tests de ces 2 classes
void demo1() {

   cout << "Demo1 : 2 classes des points :
            points entiers et points reels :\n\n";
   PointEntier A(5, 7);
   afficher(A);

   PointReel B(12.86, 21.15);
   afficher(B);

   cout << "\nFin de demo1\n\n";
}
```

```
// Pour avoir les coordonnés du point de n'importe quel type T, on définit un
// patron de classe

template <class T>
class Point {
private :
    T x, y ;
  public :

    // cas de définition directe
    Point (T a = 0, T b = 0) {
      x = a, y = b ;
    }

    // cas de définition reportée
    double longueur () const ;

    //cas des fonctions amies
    friend void afficher  (const Point<T> &) ;
    // sous linux avec g++ il faut ajouter <> après le nom de
    // la fonction sinon le compilateur va considérer la fonction
    // comme non-template :
    // friend void afficher <> (const Point<T> &) ;

};

template <class T> double Point<T>::longueur() const {
   return  sqrt( x * x + y * y);
}

template <class T> void afficher(const Point<T> & P) {
                cout << "<x = " << P.x << ", y = " << P.y
                  << ", longueur = " << P.longueur() << ">\n\n";
}
```

```
// test du patron de classe Point
void demo2() {

  cout << "Demo2 : patron de classe (1 seule classe Point) :\n\n";

  Point <int> C(5,7);
  afficher(C);

  Point <double> D(12.86, 21.15);
  afficher(D);

  cout << "\nCa marche!, fin de demo2\n\n";
}


void main() {

    demo1();
    demo2();
}

/* Exécution :
Demo1 : 2 classes des points : points entiers et points reels :

<x = 5, y = 7, longueur = 8.60233>

<x = 12.86, y = 21.15, longueur = 24.7528>


Fin de demo1

Demo2 : patron de classe (1 seule classe Point) :

<x = 5, y = 7, longueur = 8.60233>

<x = 12.86, y = 21.15, longueur = 24.7528>


Ca marche!, fin de demo2

Press any key to continue

*/
```