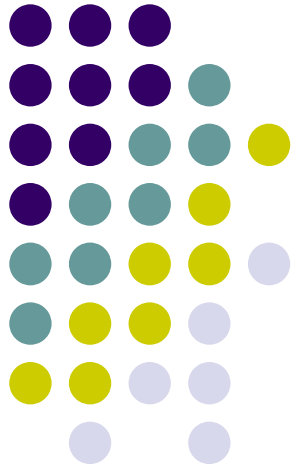
Cours Programmation Orientée Objet (en C++)

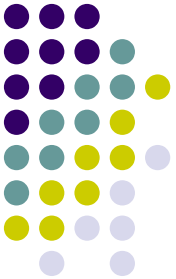
L2 Informatique

Université Cheikh Anta Diop

M. DEYE

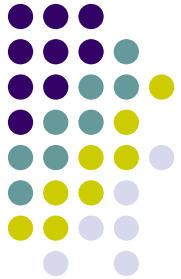


Plan du cours



- Introduction
- Qu'est ce que la POO ?
- Minimum sur les entrées sorties
- Rappel sur les références et les pointeurs
- Spécificités de C++ par rapport à C
- Classes et objets en C++
- Construction et initialisation d’objets
- Héritage
- Sur-définition d'opérateurs =, *, <<, >>, ()

Introduction

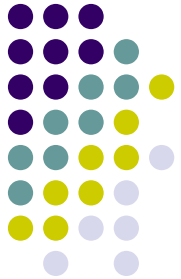


- L’histoire de la programmation a commencé avec l’invention de l’ordinateur

- la programmation, dans sa forme la plus basique, n'est rien d'autre que de donner des instructions à exécuter à l'ordinateur

- En effet, sans instructions l’ordinateur ne fera rien du tout

Introduction

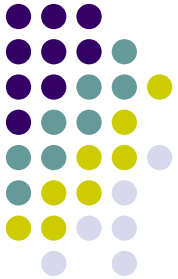


- Au début de l'informatique, la programmation se faisait uniquement avec le langage machine

- **Langage machine** : instructions spécifiques à chaque type de processeur; instructions et données sont codées en binaire (10010011…)

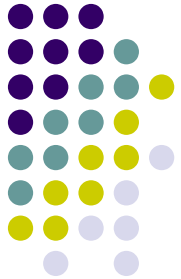
- **Langage assembleur** : ADD, MOVE, ...

Introduction



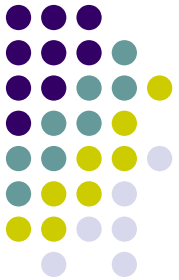
- Une nouvelle évolution a été l'arrivée de la programmation procédurale ou structurée (ex. le langage C au début des années 1970)
- **Programmation procédurale ou structurée** :
  - Le problème à résoudre est décomposé récursivement en sous-problèmes jusqu'à descendre à des actions primitives. Un programme est ainsi décomposé en un ensemble de sous-programmes appelés procédures
  - Un écart important entre les termes utilisés du coté de programmeurs et ceux utilisés par les utilisateurs pour exprimer leurs besoins

Introduction



- **Programmation procédurale ou structurée** :
  - La moindre modification des structures de données d'un programme conduit à la révision de toutes ses procédures
  - Absence de protection de données (les variables globales sont accessibles dans toutes les procédures)
  - Pour de très grosses applications, le développement peut être très long
  - L'augmentation de la taille des programmes et la nécessité de faire évoluer ces programmes

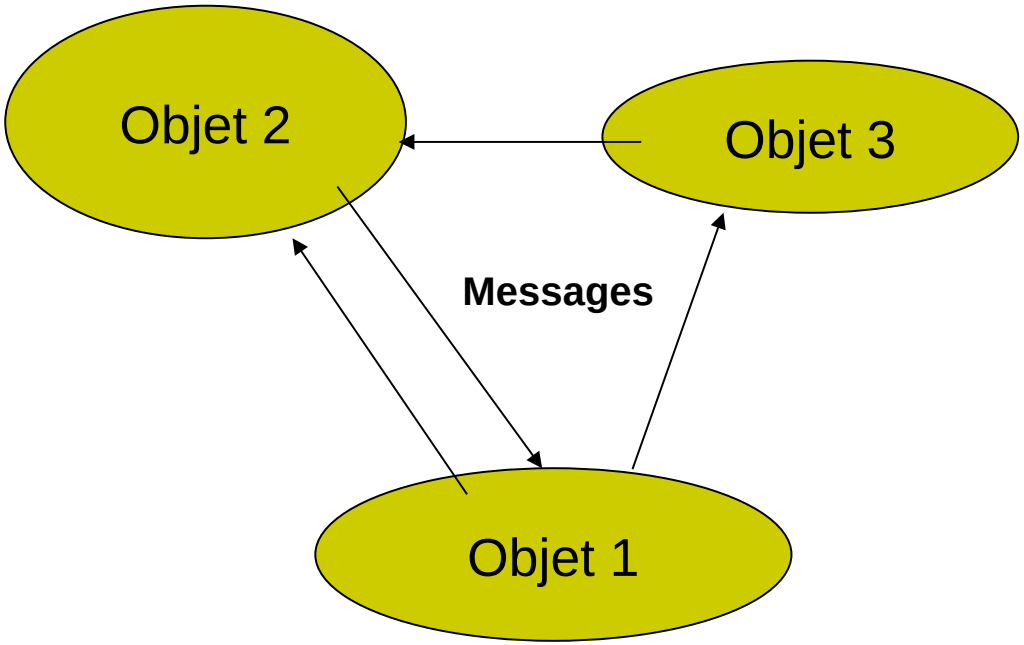
Qu'est ce que la POO ?



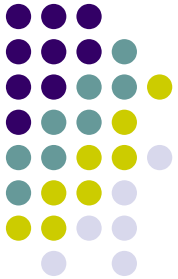
- **La programmation orientée objet** : C++, Java, Python, C#, ...
  - Le système développé est structuré par rapport aux données qu'il doit manipuler et non selon les fonctionnalités à remplir
  - Réduit l'écart entre les programmeurs et les utilisateurs finaux
  - Approche ascendante : Identifier les briques logicielles (objets) et les composer entre eux pour créer le système

Qu'est ce que la POO ?

Un Programme Orienté Objet : un ensemble d'objets autonomes et responsables qui s'entraident pour résoudre un problème final en s'envoyant des messages.

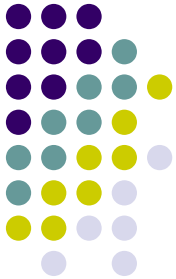


Qu'est ce que la la POO ?



- **Objectifs de la POO**:
  - Faciliter la réutilisation du code : réutiliser des fragments de code développés dans un cadre différent pour résoudre un autre problème (Programmation par composants)
  - Faciliter l'évolution du code : modifier un programme pour satisfaire à une évolution des spécifications (l'extensibilité)

Qu'est ce que la POO ?



Vocabulaire objet :

Objet

- Un objet est une entité, qui possède un état et un comportement :

  objet = état + comportement

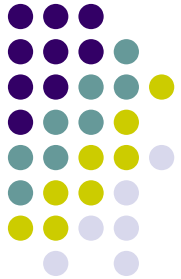
Classe

- Vue de la programmation objet, une *classe* est un type structuré de données. Nous verrons qu'une classe C++ est le prolongement des structures C (mot-clé **struct**)
- Vue de la modélisation objet, une classe correspond à un concept du domaine modélisé. Une classe regroupe des objets qui ont des propriétés et des comportements communs

Abstraction

- Considérer uniquement les attributs qui sont utiles pour le contexte étudié

Qu'est ce que la POO ?



Vocabulaire objet :

Instance

- Pour désigner un objet de la classe, on dit aussi une *instance*. « instance » est un anglicisme qui possède une signification proche de celle de « exemple » en français. On dit souvent qu'une instance « instancie » une classe. Cela signifie que l'instance est un exemple de la classe.

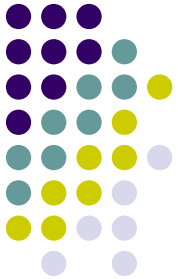
- En C++, pour désigner une instance on dit plutôt un objet.

Attribut ou membre

- L'état d'un objet est l'ensemble des valeurs de ses attributs. Un *attribut* est une propriété de l'objet.

- En C++, on dit *membre*.

Qu'est ce que la POO ?



Vocabulaire objet :

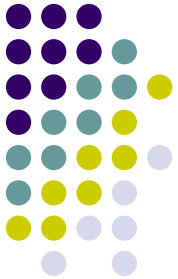
Méthode ou fonction membre

- Vue de la modélisation objet, une *méthode* est une opération que l'on peut effectuer sur un objet
- Vue de la programmation objet, une méthode est une fonction qui s'applique sur une instance de la classe.
- En C++, on dit *fonction membre*.

Message

- Les objets « communiquent » en « envoyant » et « recevant » des « messages ».
  - envoyer un message à un objet = appeler une méthode associée à l'objet
  - recevoir un message = entrer dans le corps de la méthode appelée

Qu'est ce que la POO ?



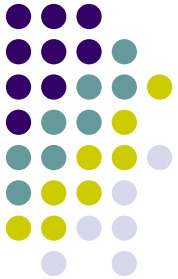
Vocabulaire objet :

Encapsulation des données

- Les données membres ne sont pas directement accessibles à l'extérieur de la classe
- L'encapsulation est appliquée de manières très diverses suivant les langages
- Le C++ prévoit les mot-clés '**private**' et '**public**' pour dire si un attribut est visible ou non de l'extérieur de la classe

- Conséquences :
  - Un objet n'est vu que par ses spécifications publiques (son interface)
  - Une modification interne est sans effet pour le fonctionnement général du programme
  - Meilleure réutilisation de l'objet

Qu'est ce que la POO ?



Vocabulaire objet :

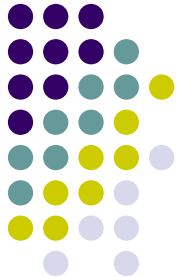
L'Héritage

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

Polymorphisme (du Grecque→ plusieurs formes)

- Un nom (de fonction, d'opérateur) peut être associé à plusieurs mais différentes utilisations
  - Surcharge ou en anglais overloading
  - Redéfinition ou en anglais overriding,
  - Généricité ou en anglais template

Minimum sur les entrées sorties



Bien sûr, C++ dispose des routines offertes par la bibliothèque standard du C ANSI <stdio.h>.

- Mais il comporte aussi des possibilités d'entrées sorties propres.
- **<iostream.h>** est la bibliothèque du C++ permettant de faire des entrées sorties

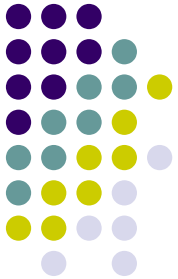
Écrire sur la sortie standard

- Pour écrire, sur la sortie standard on utilise le " stream " de sortie standard **cout**
  Là où en C, on écrivait : printf("Bonjour");
  En C++, on utilisera : cout << "Bonjour" ;

- L'opérateur **<<** prend un stream en premier opérande et une expression de type quelconque en deuxième opérande et retourne le premier opérande après avoir écrit la chaîne de caractères sur le stream.

- On peut chaîner l'utilisation de << pour écrire plusieurs chaînes de caractères à la suite les unes des autres.
  cout << "Hello " <<" World !" << endl;

Minimum sur les entrées sorties



- Créer le projet hello de type Console Application :

```
//main.cpp
#include <iostream.h>
void main()
{
    cout << "Hello World !";
    cout << "Hello World !\n";
    cout << "Hello World !" << endl;

    int n = 5;
    cout << "La valeur est " << n << endl;
    float f = 3.14f;
    char * ch = "Coucou";
    cout << ch << " float = " << f << endl;
}
```

endl manipulateur qui signifie le saut de ligne.

Minimum sur les entrées sorties

Lire sur l'entrée standard

- Le programme suivant demande d'entrer une valeur sur l'entrée standard, puis il la lit avec l'opérateur **>>** et le stream **cin** de l'entrée standard ; enfin il affiche la valeur entrée par l'utilisateur.

```
#include <iostream.h>
void main () {
    int n; float x ; char t[64] ;
    cout << " entrer un entier, un flottant et une chaîne de caractères:" ;
    cin >> n >> x >> t ;
    cout << " l'entier vaut " << n ;
    cout << " le flottant vaut " << x ;
    cout << " la chaîne vaut " << t << endl ;
}
```

- De même que pour <<, l'opérateur >> accepte presque tous les types et il est associatif.

Minimum sur les entrées sorties

- Les avantages des flux C++ sont nombreux, on notera en particulier ceux-ci :
  - le type des donnée est automatiquement pris en compte par les opérateurs d'insertion et d'extraction (ils sont surchargés pour tous les types prédéfinis) ;
  - les opérateurs d'extraction travaillent par référence (on ne risque plus d'omettre l'opérateur & dans la fonction scanf) ;
  - il est possible de définir des opérateurs d'insertion et d'extraction pour d'autres types de données que les types de base du langage ;
  - leur utilisation est globalement plus simple.

Rappel sur les références et les pointeurs

- Quand on déclare une variable avec un nom et un type, un emplacement mémoire du type de la variable est créé à une certaine adresse avec son nom pour y accéder.

- L'emplacement mémoire recevra la valeur de la variable lors d'une affectation.

```
int x ; // une déclaration

x = 3 ; // une affectation
```

Rappel sur les références et les pointeurs

- Le C permet de manipuler dans le programme, les adresses des emplacements mémoire des variables
- &x désigne l'adresse de la variable x
- On peut déclarer des *pointeurs* d'un type qui sont des variables contenant des adresses de variables de ce type avec le symbole *.

```
int  x ; // une déclaration
x = 3 ; // une affectation
int *  p ; // un pointeur sur un entier
p = &x ; // p vaut l'adresse de x
```

Rappel sur les références et les pointeurs

- L'opérateur * s'appelle l'opérateur de *déréférencement* et *p désigne la valeur contenue dans l'emplacement mémoire dont l'adresse et p

```
int  x ; // une déclaration
x = 3 ; // une affectation
int *  p ; // un pointeur sur un entier
p = &x ; // p vaut l'adresse de x
int y = *p ; // y vaut 3
```

Rappel sur les références et les pointeurs

- On peut aussi déclarer une *référence* à une variable existante
- Une référence se déclare avec l'opérateur &
- Une référence est un synonyme - un alias (un nom supplémentaire) – pour désigner l'emplacement mémoire d'une variable.

```
int &  z = x ; // z est une référence a x, z vaut 3
```

Rappel sur les références et les pointeurs

- Noter que les opérateurs & et * sont inverses l'un de l'autre. On a toujours :

  *(&x) = x et &(*p) = p

- Attention, on ne peut pas déréférencer un pointeur qui ne contient pas une adresse valide :

  ```
  int * q ;
  *q = 7 ; // plantage
  ```

- Il faut initialiser le pointeur avant :

  ```
  int * q = &x;
  *q = 7 ; // ok
  ```

Rappel sur les références et les pointeurs

- Si on change la valeur de x, la valeur de z est aussi changée et inversement. Idem avec *p.

```
int  x ; // une déclaration
x = 3 ; // une affectation
int *  p ; // un pointeur sur un entier
p = &x ; // p vaut l'adresse de x
int y = *p ; // y vaut 3
int &  z = x ; // z est une référence a x, z vaut 3
x = 4; // x et z valent 4, *p aussi
z = 5; // x et z valent 5, *p aussi
*p = 6; // *p, x et z valent 6
```

Rappel sur les références et les pointeurs

- Noter que les opérateurs & et * sont inverses l'un de l'autre. On a toujours :

  *(&x) = x et &(*p) = p

- Attention, on ne peut pas déréférencer un pointeur qui ne contient pas une adresse valide :

  ```
  int * q ;
  *q = 7 ; // plantage
  ```

- Il faut initialiser le pointeur avant :

  ```
  int * q = &x;
  *q = 7 ; // ok
  ```

Rappel sur les références et les pointeurs

```
#include <iostream>
using namespace std;
int main(){
   int  x ; // une déclaration
   x = 3 ; // une affectation
   int *  p ; // un pointeur sur un entier
   p = &x ; // p vaut l'adresse de x
   int y = *p ; // y vaut 3
   Int  &  z = x ; // z est une référence a x, z vaut 3
   cout << "x = "<<x<<" *p = "<<*p<<" y= "<<y<<" z = "<<z<<endl;
   x = 4; // x et z valent 4, *p aussi
   cout << "x = "<<x<<" *p = "<<*p<<" y= "<<y<<" z = "<<z<<endl;
   z = 5; // x et z valent 5, *p aussi
   cout << "x = "<<x<<" *p = "<<*p<<" y= "<<y<<" z = "<<z<<endl;
   *p = 6; // *p, x et z valent 6
   cout << "x = "<<x<<" *p = "<<*p<<" y= "<<y<<" z = "<<z<<endl;
   return 0;
}
```

```
x= 3 *p= 3 y= 3 z= 3
x= 4 *p= 4 y= 3 z= 4
x= 5 *p= 5 y= 3 z= 5
x= 6 *p= 6 y= 3 z= 6
```

Spécificités de C++ /C

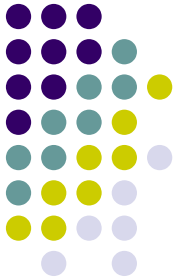
- C++ dispose d'un certain nombre de spécificités par rapport a C qui ne sont pas axées sur l'orienté objet :
  - l'emplacement libre des déclarations
  - le passage de paramètres par référence
  - les arguments par défaut
  - la sur-définition de fonction
  - les opérateurs new et delete
  - Les espaces de noms

Emplacement des déclarations

- L'emplacement des déclarations est *libre* en C++. Par contre, on ne peut utiliser la variable déclarée que dans les instructions du bloc où est effectuée la déclaration et postérieures à la déclaration.

```
void f() {
    i = 4 ; // incorrect
    int i ; // i est déclaré ici
    i = 4 ; // correct
        {
            float f ; // f est declare ici
            f = 4.0 ; // correct
        } // fin de la portee de f
    f = 4.0 ; // incorrect
    i = 5 ; // correct
} // fin de la portee de i
```

Arguments par référence



- En C, les arguments et la valeur de retour d'une fonction sont transmis par valeur. Pour simuler en quelque sorte ce qui se nomme '' transmission par adresse'' dans d'autres langages il est nécessaire de ''jongler'' avec les pointeurs.

Transmission des arguments par valeur

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

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

Arguments par référence

Transmission des arguments par adresse

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

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

Arguments par référence

- C++ permet de demander au compilateur de prendre lui-même en charge la transmission des arguments par adresse

Transmission des arguments par référence

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

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

Arguments par défaut

- En C il est indispensable que l'appel de la fonction contienne exactement le même nombre et type d'arguments que dans la déclaration de la fonction. C++ permet de s'affranchir de cette contrainte en permettant l'usage d'arguments par défaut.

```
void f(int, int = 12) ;
void main () {
    int n = 10 ; int p = 20 ;
    f(n, p) ;
    f(n) ;
}
void f(int a, int b) {
    cout << " premier argument : " << a;
    cout << " second argument : " << b <<"\n";
}
```

```
premier argument : 10 second argument : 20
premier argument : 10 second argument : 12
```

Arguments par défaut

- Lors d'une déclaration avec des arguments par défaut, ceux-ci doivent être les derniers de la liste des arguments.
- Le programmeur doit fixer les valeurs par défaut dans la déclaration de la fonction.

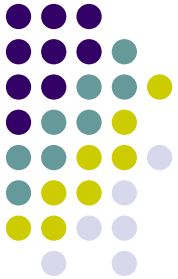
```
void f(int = 0, int = 12) ; // Définition de valeurs par défaut
void main () {                //pour plusieurs arguments
    int n = 10 ; int p = 20 ;
    f(n, p) ;
    f(n) ;
    f();
}
void f(int a, int b) {
    cout << " premier argument : " << a;
    cout << " second argument : " << b <<"\n";
}
```

Surdéfinition de fonction

- Un même identificateur peut désigner plusieurs fonctions, à conditions qu'elles diffèrent par la liste des types de leurs arguments(Surcharge ou en anglais overloading).

```
void sosie(int) ;
void sosie(double) ;
main () {
    int n = 10 ; double x = 4.0 ;
    sosie(n) ;
    sosie(x) ;
}
void sosie(int a) {
    cout << " sosie avec INT : " << a << endl ;
}
void sosie(double b) {
    cout << " sosie avec DOUBLE : " << b << endl ;
}
```

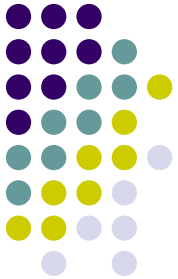
Surdéfinition de fonction



- Écrivez la fonction maxi?

```
void main() {
        float x, y, z;
        float T[] ={11.1, 22.2, 33.3, 44.4, 7.7, 8.8 };
        x = maxi(1.86, 3.14);
        y = maxi(1.86, 3.14, 37.2);
        z = maxi(6, T);
        cout << x <<" "<< y <<" "<< z;
}
```

Surdéfinition de fonction



```
float maxi(float a, float b) {
        return (a > b) ? a : b;
}

float maxi(float a, float b, float c) {
        return maxi(a, max(b, c));
}

float maxi(int n, float t[]) {
        if (!n) return 0;
        float m = t[1];
        for (int i = 2 ; i < n; i++)
            m = maxi(m, t[i]);
        return m;
}
```

Opérateurs new et delete

- En C, la gestion dynamique de la mémoire fait appel aux fonctions malloc et free. En C++, on utilise les fonctions new et delete.

Exemple 1

```
int * ad ;
```

en C++ on alloue dynamiquement comme cela :

```
ad = new int ;
```

alors qu'en C il fallait faire comme ceci :

```
ad = (int *) malloc (sizeof(int)) ;
```

Exemple 2

```
char * adc;
```

L'instruction : adc = new char[100]; alloue l'emplacement nécessaire pour un tableau de 100 caractères et place l'adresse (de début) dans adc.
En C le même résultat : adc=(char *) malloc(100);

Opérateurs new et delete

Plus généralement, si on a un type donné **type**, on alloue une variable avec :

```
new type ;
```

ou un tableau de n variables avec :

```
new type [n] ;
```

On désalloue dynamiquement de la mémoire (allouée avec new) comme cela :

```
delete ad;
```

Plus généralement, on désalloue une variable x (allouée avec new) avec :

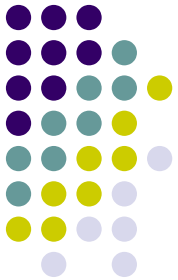
```
delete x;
```

ou un tableau de variables (allouée avec new []) avec :

```
delete [] x ;
```

En C++, bien que l'utilisation de malloc et free soit toujours permise, il est très conseillé de n'utiliser que new et delete.

Les espaces de noms



- **Un espace de nom (namespace) est une zone de déclaration d'identificateurs permettant au compilateur de résoudre les conflits de noms**

```
namespace first_space{
   int N=20 ;
   void func(){
      cout << "Inside first_space" << endl;
   }
}
```

```
namespace second_space{
   int N=30 ;
   void func(){
      cout << "Inside second_space" <<endl;
   } }
```

Les espaces de noms

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

//declaration de namespaces ici

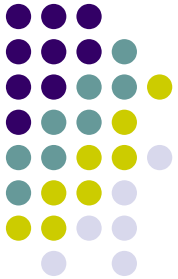
int main () {
   first_space::func();
   second_space::func();
   cout <<"N of first_space = "<<first_space::N<<endl;
   cout <<"N of second_space = "<<second_space::N<<endl;
  return 0;
}
```

Les espaces de noms

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

namespace first_space{
   int N=20;
   void func(){
      cout << "Inside first_space" << endl;
   }}
namespace second_space{
   int N=30;
   void func(){
      cout << "Inside second_space" << endl;
   }}
using namespace first_space;
int main (){
   func();
   cout <<"N = "<<N<<endl;
   return 0;
}
```

Classes et objets en C++



- Déclaration de classes
- Définition des fonctions membres
- Membres privés ou publiques
- Affectation d’objets
- Constructeurs
- Destructeur
- Règles d'utilisation des fichiers .cpp et .h
- Membres statiques

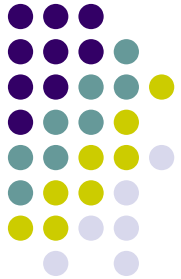
Déclaration de classes

- La déclaration d'une classe est voisine de celle d'une structure. En effet, il suffit:
  - De remplacer le mot clé struct par le mot clé class,
  - De préciser quels sont les membres publics(fonctions ou données) et les membres privés en utilisant les mots public et private.

```
class Point {
    int x ;                      // un membre
    int y ;                      // un autre membre
    public :
    void initialise(int, int) ;  // une fonction membre
    void deplace(int, int);      // encore une
    void affiche() ;             // encore une fonction membre
};
```

- x et y sont des membres de la classe Point.
- initialise, deplace et affiche sont des fonctions membres de la classe Point.
- Cet exemple suit le principe d'encapsulation?

Définition des fonctions membres



- La définition d'une fonction membre suit la syntaxe de la définition d'une fonction C avec un nom préfixé par le nom de la classe et quatre points '::'.
- Le symbole '::' s 'appelle l'opérateur de résolution de portée.

```
void Point::initialise(int a, int b) {
    x = a;
    y = b;
}
void Point::deplace(int dx, int dy) {
    x += dx;
    y += dy;
}
void Point::affiche() {
    cout << " x = " << x << endl;
    cout << " y = " << y << endl;
}
```

x et y sont visibles des fonctions membres de la classe Point.

Exemple d'utilisation de la classe Point

```
main() {
    Point a, b ;
    a.initialise(5, 2) ;
    a.affiche() ;
    a.deplace(8, 4) ;
    a.affiche() ;
    b.initialise(-1, 1) ;
    b.affiche() ;
}
```

- Pour appeler la fonction membre initialise sur l'objet a, noter qu'il faut écrire le nom de l'objet (a), un point (.) et le nom de la fonction membre (initialise).

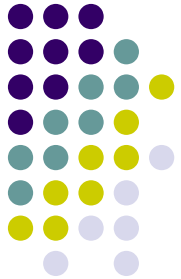
Exemple 1

```
class Point {
        int x ; // un membre
        int y ; // un autre membre
        public :
        void initialise(int, int) ; // une fonction
        membre
        void deplace(int, int); // encore une
        void affiche() ; // encore une fonction
        membre
};
void Point::initialise(int a, int b) {
        x = a;
        y = b;
}
void Point::deplace(int dx, int dy) {
        x += dx;
        y += dy;
}
```

```
void Point::affiche() {
        cout << " x = " <<
        x << endl;
        cout << " y = " <<
        y << endl;
}
main() {
        Point a, b ;
        a.initialise(5, 2) ;
        a.affiche() ;
        a.deplace(8, 4) ;
        a.affiche() ;
        b.initialise(-1, 1) ;
        b.affiche() ;
}
```

a.x =3 ; dans le main() serait rejetée à la compilation?

Membres privés ou publiques



- La déclaration d'une classe est toujours du type :

```
class Toto {
... // des membres prives
public :
... // des membres publiques
private :
... // encore des membres prives
public :
... // encore des membres publiques
... // etc.
};
```

- Si on omet le mot-clé public dans une classe, aucun de ses membres n'est accessible de l'extérieur de la classe...

Constructeurs

- Il est souvent nécessaire d'initialiser les objets au moment de leur création.

- Dans le cas de la classe Point, on souhaite pouvoir initialiser les membres x et y. Dans d'autres cas, ce peut être pour effectuer une allocation mémoire, etc.

- Une solution pourrait être de définir pour toutes ces classes une méthode initialise qui réaliserait les initialisations souhaitées.

- Mais cela est problématique :
- Pour toute création d'un nouvel objet, deux actions vont être nécessaires (déclaration + appel de la méthode initialise) alors que la création d'un objet est *a priori* une action atomique.
- Et que faire si un client oublie d'appliquer la méthode initialise?

Constructeurs

- Pour résoudre ce problème, C++ possède un mécanisme d'initialisation automatique d'objets de classe.
- Une ou plusieurs méthodes particulières, appelées *constructeurs*, sont appliquées implicitement dès qu'un objet est défini.
- Ces constructeurs, généralement publics, portant le même nom que la classe à laquelle ils appartiennent.
- Les constructeurs n'ont aucun type de retour (même pas void) et ne sont jamais appelés explicitement par le programmeur.
- C'est le compilateur qui se charge de le faire à chaque création d'objet, après avoir choisit le constructeur à utiliser en fonction des paramètres d'initialisation fournis (principe de la surcharge).

Destructeur

- Le langage C++ offre un mécanisme complémentaire au constructeur - Une fonction membre particulière pour désinitialiser les objets :le **destructeur**.

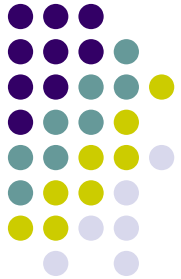
- Elle est invoquée systématiquement lorsqu' un objet doit être détruit.

- De même que le constructeur, le destructeur est une fonction membre sans type de retour.

- Il porte le nom de la classe précédé d'un tilde '~'.

- Un destructeur ne possède jamais d'argument; par conséquent ?

Constructeur et destructeur d’objets



- Le but d’un constructeur est :
  - d’allouer un emplacement mémoire pour l’objet,
  - d’initialiser les membres de l’objet avec de bonnes valeurs de départ et
  - de retourner l’adresse de l’emplacement mémoire choisi.

- Un destructeur d’objet :
  - remet l’objet dans un état terminal et
  - libère l’emplacement mémoire associé à l’objet.

Constructeur et destructeur d'objets

- En fait, notre exemple précédent était approximatif et en bon C++, il faut écrire :

```
class Point {
    ...
    public :
    Point(int, int) ; // le constructeur de la classe
    ~Point() ; // le destructeur de la classe
    ...
};
Point : : Point(int a, int b) {
    x = a ;
    y = b ;
}
Point : : ~Point() {} // le destructeur ne sert pas dans cet exemple
```

Constructeur et destructeur d'objets

- A partir du moment où un constructeur existe, il n'est pas possible de créer un objet sans fournir les arguments requis par le constructeur.

- Par exemple, la classe Point comporte le constructeur de prototype
  Point(int, int)

- Les déclarations suivantes seront incorrectes :
  Point a; //incorrect : le constructeur attend deux arguments
  Point(4) //incorrect : (même raison)

- Celle-ci, en revanche, conviendra :
  Point a(5, 2) ; //correct car le constructeur possède deux arguments

Pour comprendre les moments où sont appelés les constructeurs et destructeurs d'objets?

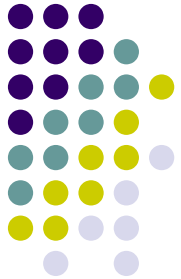
```
class Test {
      int num ;
      public :
      Test(int) ;     //déclaration constructeur
      ~Test() ;     //déclaration destructeur
} ;
Test : :Test(int n) {    //définition constructeur
      num = n ;
      cout << "++  Appel constructeur – num = " << num <<
            endl ;
}
Test : :~Test() {
      cout << "-- Appel destructeur – num =  " << num <<
            endl ;
}
void fct(int p) {
      Test x(2*p) ;
}
main() {
      Test a(1) ;
      fct(1) ;
      fct(2) ;
}
```

La sortie du programme est :

```
++  Appel constructeur - num = 1
++  Appel constructeur - num = 2
-- Appel destructeur - num =  2
++  Appel constructeur - num = 4
-- Appel destructeur - num =  4
-- Appel destructeur - num =  1
Press any key to continue
```

a est détruit à la sortie du programme main et les objets x sont détruits à la sortie de fct.

Accesseurs (getters) et mutateurs (setters)



La notion d'accesseur ou getter :

- Un getter est une fonction membre publique permettant de récupérer la valeur d'une donnée membre protégée (private)

- Il doit avoir comme type de retour le type de la donné membre et ne possède pas nécessairement d'arguments

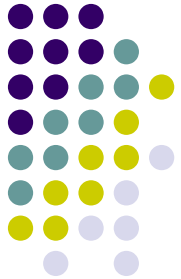
- Une convention de nommage veut que son nom commence de façon préferrentielle par le préfixe *Get*, afin de faire ressortir sa fonction première

Accesseurs (getters) et mutateurs (setters)

```
class Personne{
        private :
                int  _age;
        public :
                int GetAge();
};

int Personne::GetAge(){
                return  _age;
}
```

Accesseurs (getters) et mutateurs (setters)



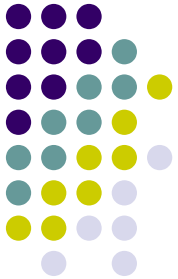
La notion de mutateur ou setter :

- Un setter est une fonction membre publique permettant de modifier une donnée membre protégée(private)

- Il doit avoir comme argument la valeur à assigner à la donnée membre et ne doit pas nécessairement renvoyer de valeur (renvoie void le plus souvent)

- Une convention de nommage veut que son nom commence de façon préférentielle par le préfixe Set

Accesseurs (getters) et mutateurs (setters)



```
class Personne{
        private :
                int  _age;
        public :
                void SetAge( int);
};

void Personne::SetAge(int age){
                if (age<150)  _age=age;
}
```

Accesseurs (getters) et mutateurs (setters)

```
class Personne{
        private :
                int  _age;
        Public :
                int GetAge() ;
                void SetAge( int);
};
int Personne::GetAge(){
        return  _age;
}
void Personne::SetAge(int age){

        if (age<150)   _age=age;
}
```

Règles d'utilisation des fichiers .cpp et .h

- Jusqu'ici, nous avions regroupé au sein d'un même programme trois sortes d'instructions destinées à :
  - La déclaration de la classe,
  - La définition de la classe,
  - L'utilisation de la classe.

- Pour pouvoir effectivement réutiliser les classes programmées et les compiler séparément, il est indispensable de les exploiter proprement et les ranger dans des fichiers qui portent des noms corrects.

  - La déclaration d'une classe **Classe** doit être mise dans un fichier **classe.h**
  - La définition d'une classe **Classe** doit être mise dans un fichier **classe.cpp**

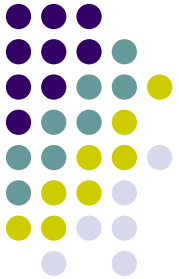
Règles d'utilisation des fichiers .cpp et .h

- Pour l'instant, retenir qu'un fichier classe.h possède la structure suivante :

```
// fichier classe.h
#ifndef CLASSE_H
#define CLASSE_H
#include ... // includes de classes eventuelles
...
class Classe {
...
} ;
#endif
```

- **#ifndef #define** et **#endif** servent de garde-fou pour que le fichier ne soit effectivement inclus qu'une seule fois lors d'une compilation.

Règles d'utilisation des fichiers .cpp et .h



- Et qu'un fichier classe.cpp possède la structure suivante :

```
// fichier classe.cpp
#include "classe.h " // include de sa propre classe
#include ... // autres includes optionnels
...
Classe : :Classe(...) { // definition du constructeur
...
}
Classe : :~Classe() { // definition du destructeur
...
}
... // autres definitions de fonctions membres
```

- Noter l'include obligatoire de classe.h; les autres sont optionnels.

Membres statiques

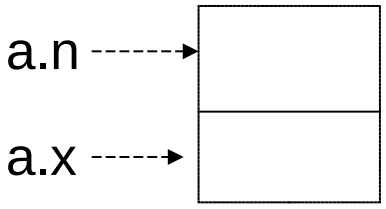
- Avec le qualificatif '**static**' avant un membre d'une classe on peut spécifier qu'un membre est commun à tous les objets de la classe.

- En modélisation objet, cela correspond aux attributs « de classe ». Un membre sans le mot-clé **static** correspond à un attribut « d'instance ».

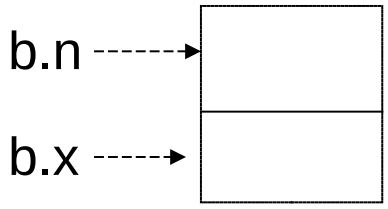
```
class Exemple
{
      int n ;
      float x ;
      ...
} ;
```

Une déclaration telle que **Exemple a,b;**

Conduit à une situation que l'on peut schématiser ainsi



Objet a



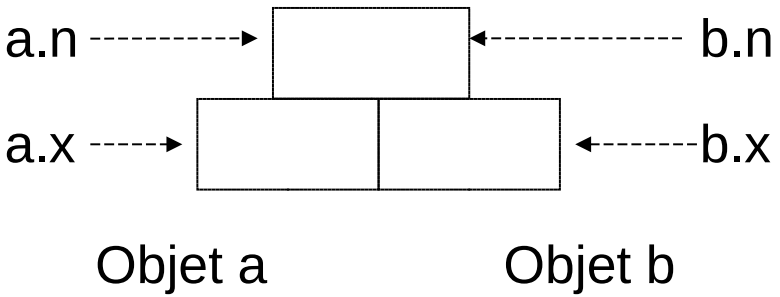
Objet b

Membres statiques

```
class Exemple {
      static int n ;
      float x ;
      ...
} ;
```

Une déclaration telle que **Exemple a,b;**

Conduit à une situation que l'on peut schématiser ainsi



Les membres données statiques sont des sortes de variables globales dont la portée est limitée à la classe.

Membres statiques

- Les membres données statiques n'existent qu'en un seul exemplaire, indépendamment des objets de la classe(même si aucun objet n'a encore été créé).

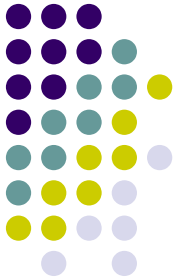
- Leur initialisation ne peut plus être fait par le constructeur.

```
class Exemple {
        static int n = 2; //erreur
        float x ;
        ...
} ;
```

Un membre statique doit être initialisé dans le .cpp tout comme une fonction membre sinon une erreur sera produite à l'édition de liens:

```
int Exemple : :n = 2 ;
```

Membres statiques



Écrire une classe compteur permettant d'afficher à tout moment le nombre d'objet existants?

```
++ constructeur : il y a maintenant  1 objets
++ constructeur : il y a maintenant  2 objets
++ constructeur : il y a maintenant  3 objets
++ destructeur : il reste maintenant  2 objets
++ destructeur : il reste maintenant  1 objets
++ constructeur : il y a maintenant  2 objets
++ destructeur : il reste maintenant  1 objets
++ destructeur : il reste maintenant  0 objets
Press any key to continue
```

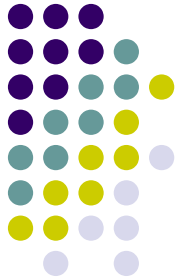
Afficher à tout moment le nombre d'objet existants :

```
//compteur.h
#ifndef COMPTEUR_H
#define COMPTEUR_H
class compteur {
    static int nbre;
    public :
    compteur();
    ~compteur ();
};
#endif
```

```
//compteur.cpp
#include <iostream.h>
#include « compteur.h "

int compteur::nbre =0;

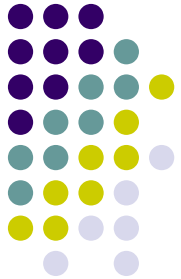
compteur::compteur ()
{cout<<"++ constructeur : il y a
maintenant  "<< ++nbre<<" objets\
n";
}
compteur::~compteur ()
{cout<<"++ destructeur : il reste
maintenant  "<< -- nbre <<" objets\
n";
}
```



Afficher à tout moment le nombre d'objet existants :

- ```
  //User.cpp
  #include <iostream.h>
  #include « compteur.h "
  main(){
      void fct();
      compteur a;
      fct();
      compteur b;
  }
  void fct(){
      compteur u, v;
  }
  ```
  ```

Propriétés des fonctions membres



- Les améliorations concernant les fonctions restent valables aussi pour les fonctions membres :
  - surdéfinition de fonctions membres,
  - arguments par défaut,
  - objets transmis en argument d'une fonction membre, par valeur, adresse ou référence.

- Les améliorations propres aux fonctions membres :
  - fonctions membres en ligne,
  - fonctions membres statiques,
  - Autoréférence: le mot-clé this,

Fonctions membres en ligne

- Au lieu de mettre le mot-clef “ **inline**” devant la définition de la fonction, on écrit le corps de la fonction au même endroit que sa déclaration dans la classe.

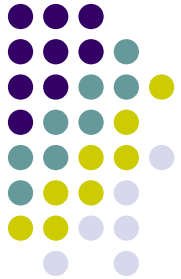
- Ci-dessous la fonction membre deplace de la classe Point est normale :

```
class Point { …
    void deplace(int, int);
};
void Point::deplace(int dx, int dy) { x += dx; y += dy; };
```

- Ci-dessous elle est inline bien que le mot-clef ne soit pas présent.

```
class Point { …
    void deplace(int dx, int dy) { x += dx; y += dy; };
};
```

Fonctions membres statiques



- Une fonction membre statique C++ correspond à une méthode de classe dans le vocabulaire objet.

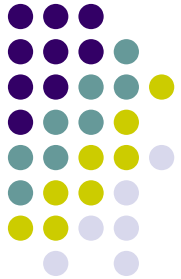
- Une fonction membre non statique C++ correspond à une méthode d'instance dans le vocabulaire objet.

- Ci-dessous, la fonction membre **affiche_tout** de la classe Point est statique et la fonction membre **affiche** est normale.

```
class Point { …
    void affiche();
    static void affiche_tout();
};
…
Point::affiche_tout();
```

- Une fonction membre statique s'appelle avec son nom préfixé du nom de la classe et de **::**

Autoréférence: le mot-clé *this*



- Quand le programme est dans une fonction membre d'instance, le programmeur peut manipuler l'adresse de l'instance courante avec le mot-clef '**this**'.

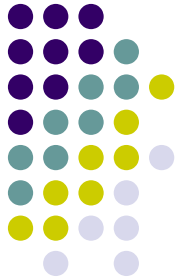
- En effet, this désigne l'adresse de l'instance courante.

```
class Point { …
    void affiche() { cout << " mon adresse est " << this << endl; }
};
```

- this est utile, par exemple, dans un constructeur (ou un destructeur) pour stocker l'adresse de l'objet construit (ou effacer l'adresse de l'objet détruit) d'un tableau ou liste d'instances ou d'un attribut d'un autre objet.

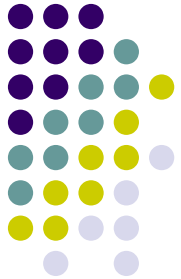
- this n'a évidemment pas de sens dans une fonction membre de classe.

Construction, destruction et initialisation des objets



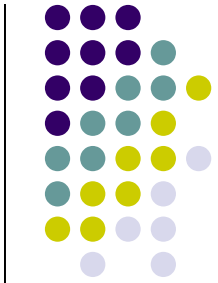
- Les objets automatiques
- Les objets statiques
- Les objets dynamiques
- Initialisation d'un objet lors de sa déclaration
- Constructeur par recopie
- Objets membres
- Initialisation de membres dans l’en-tête d’un constructeur

Construction, destruction et initialisation des objets



- En C, une variable peut être créée de deux façons :
- Par une déclaration :elle est alors de classe **automatique** ou **statique**; sa durée de vie est parfaitement définie par la nature et l'emplacement de sa déclaration
- En faisant appel à des fonctions de gestion dynamique de la mémoire(malloc, free…): elle est alors **dynamique**; sa durée de vie est contrôlée par le programme
- En C++, on retrouvera ces trois classes à la fois pour les variables ordinaires et pour les objets, avec cette différence que la gestion dynamique fera appel aux opérateurs *new* et *delete*

Les objets automatiques

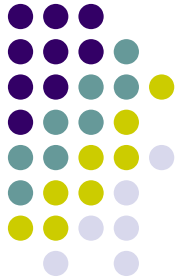


- Les objets automatiques sont les objets déclarés dans une fonction ou dans un bloc.

```
void f() {
    Truc t ; // t est construit ici
    …
    {
        Bidule b ; // b est construit ici
        …
    } // b est détruit ici
} // t est détruit ici
```

- t et b sont des objets automatiques. t est visible dans la fonction f et b est dans le bloc. A la sortie de la fonction t est détruit. A la sortie du bloc, b est détruit.

Les objets statiques

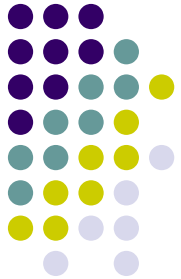


- Un objet statique est un objet déclaré avec le mot-clé static dans une déclaration de classe ou dans une fonction
- ou bien à l'extérieur de toute fonction.

- Un objet statique est créé avant le début de l'exécution du programme et il est détruit à la sortie du programme.

static Point a(1, 7);

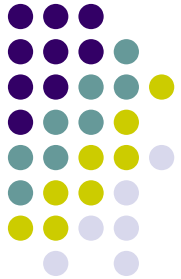
Les objets dynamiques



- Ce sont eux qui font tout l'intérêt de la programmation orienté objet. Un objet dynamique est un objet créé avec *new* et éventuellement détruit avec *delete*.

```
class point {
        int x,y;
        public :
        point(int abs, int ord) { x=abs; y=ord;
                                  cout << "++ appel constructeur" << endl; }
        ~point() { cout << "-- appel destructeur" << endl; }
}
main () {
        point * adr;
        cout << "&& debut main" << endl;
        adr = new point(3, 7);
        fct(adr);
        cout << "&& fin main" << endl;
}
void fct (point * adp) {
        cout << "&& debut fct" << endl;
        delete adp;
        cout << "&& fin fct" << endl;
}
```

Initialisation d'un objet lors de sa déclaration



- En C, on peut initialiser une variable au moment de sa déclaration, comme dans :
  int n=12;

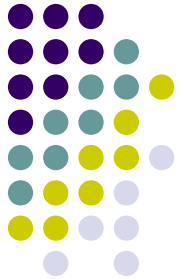
- En théorie, C++ permet de faire le même avec les objets, en ajoutant un initialiseur lors de leur déclaration.
- Avec la déclaration de la classe suivante :

```
class point {
    int x ; int y ;
    public :
    point(int abs) { x = abs ; y = 0 ; }
    ...
} ;
```

- Les deux déclarations :
  point a(3); et **point a = 3;** sont équivalentes.

- En effet, il s'agit de fournir sous une forme peu naturelle des arguments pour un constructeur.

Initialisation d'un objet lors de sa déclaration



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

- Avec la déclaration de la classe suivante :

```
struct paire { int n ; int p ; } ;
class Point {
      ...
     Point(paire q) { x = q.n ; y = q.p; }
     ...
} ;
```

- on peut écrire :

```
paire s = { 3, 8 } ;
point a(s);
```

Ou

```
point a = s;
```

Constructeur par recopie

- Un constructeur par recopie est un constructeur dont la signature est :

  NomClasse::NomClasse(NomClasse &)

- Son rôle est d'initialiser un objet par recopie d'une autre instance.

- Ainsi il est possible de déclarer des objets dont la valeur initiale est une copie d'un objet créé précédemment :

  Point p1(3,4);

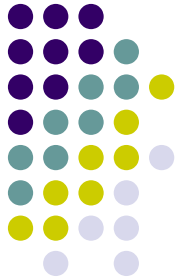
  Point p2(p1);

  Point p3=p1;

- Les points **p2** et **p3** sont initialisés par recopie du point **p1**.

- Pour la classe **Point** le constructeur par recopie est : **Point(Point &);**

Constructeur par recopie



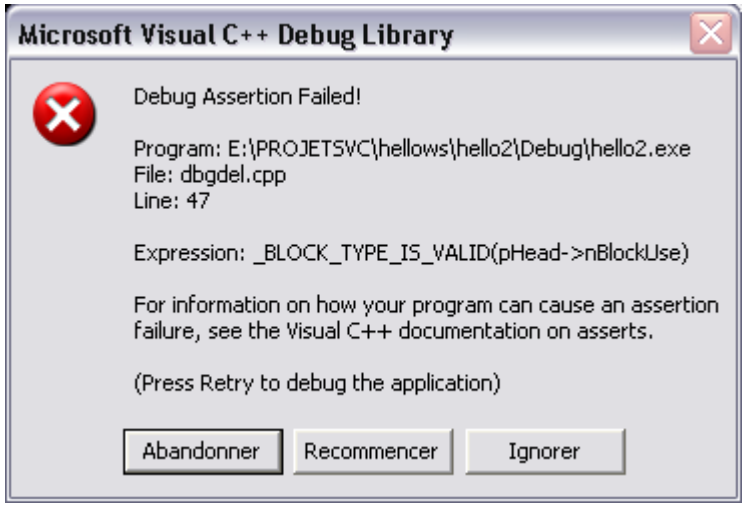
- Si le programmeur ne le définit pas, alors il existe une définition implicite :
- Les valeurs des membres de l'objet sont recopiées, une à une, telles quelles.
- Cette approche est risquée. Elle ne marche que si les attributs de l'objet sont des valeurs et pas des pointeurs.
- Elle ne marche pas lorsque l'objet recopié pointe vers d'autres objets ou structures.

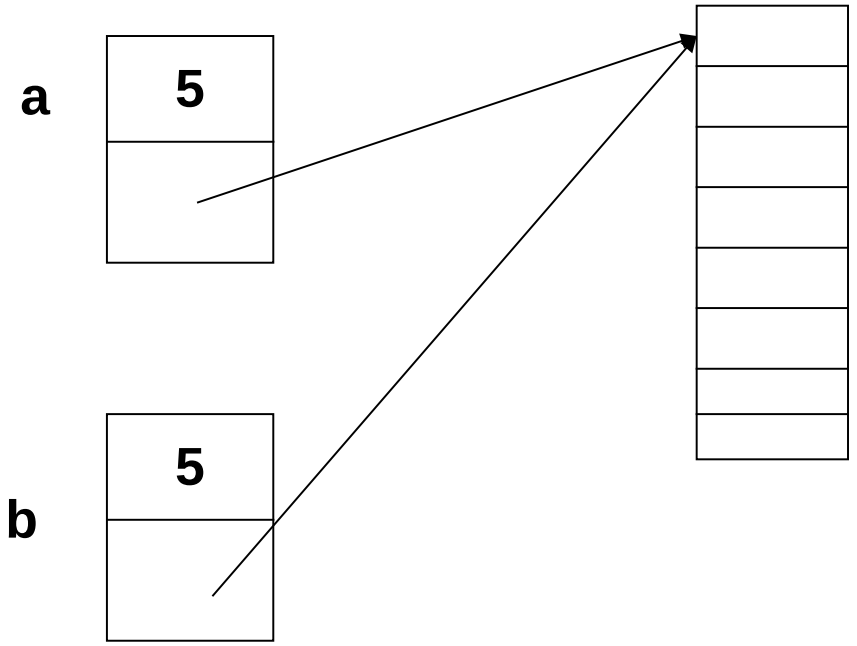
Exemple (qui ne marche pas) :

```
class vecteur {
   int nelem ;
   double * adr ;
  public :
   vecteur (int n) { adr = new double [nelem = n] ;
      cout << "+++ constructeur usuel +++ adr objet : " << this
          << " +++ adr vecteur : " << adr << "\n" ;  }
~vecteur ()
    { cout << "--- Destructeur objet --- adr objet : "
          << this << " --- adr vecteur : " << adr << "\n" ;
      delete [ ] adr ;
    }
} ;
main() {
     vecteur a(5);
     vecteur b(a);   // ou bien vecteur b=a ;
}
```

Exemple (qui ne marche pas) :

- Lors de la destruction de a, le destructeur de la classe vecteur détruit le tableau pointé par adr;
- Lors de la destruction de b, même Chose;
- Conclusion, le programme se peut se planter lors de la deuxième destruction.





Constructeur par recopie

- On peut éviter ce problème avec le constructeur de recopie suivant:

```
vecteur::vecteur(vecteur & v) {
        adr = new double [nelem = v.nelem] ;
        for (int i=0; i<nelem; i++)
            adr[i] = v.adr[i];
        cout << "+++ constructeur recopie +++ adr objet : " << this
        <<" +++ adr vecteur : " << adr <<endl;
}
```

```
+++ constructeur usuel +++ adr objet : 0x0012FF6C +++ adr vecteur : 0x00491C70
+++ constructeur recopie +++ adr objet : 0x0012FF64 +++ adr vecteur : 0x00491AA0

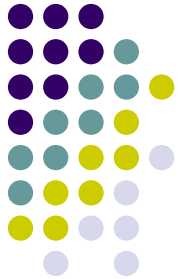
--- Destructeur objet --- adr objet : 0x0012FF64 --- adr vecteur : 0x00491AA0
--- Destructeur objet --- adr objet : 0x0012FF6C --- adr vecteur : 0x00491C70
Press any key to continue
```

Constructeur par recopie

- Les constructeurs ordinaires sont invoqués aux moments de création des objets
- Par contre le constructeur de recopie est invoqué quand un objet va être dupliquer pour créer un nouvel objet
  - Lors d'une initialisation avec l'opérateur **=** (**vecteur b=a**)
  - Lors de sa transmission par valeur (comme argument ou valeur de retour d'une fonction)
  -

```
void f(vecteur v){}
int main(){
        vecteur a(5) ;
        f(a) ;          //invoque implicitment le constructeur de recopie
        return 0;
}
```

Affectation d’objets



- On peut affecter un objet dans un autre :
  a = b ;

- Dans ce cas, les membres de b sont recopiés dans ceux de a.

- Cette recopie est insuffisante dès qu’un objet comportera des pointeurs sur des emplacements dynamiques.

- Solution surdéfinition de l’opérateur =.

Opérateur =

- Cependant il reste encore un autre problème !!
- Tester l'exécution de votre projet avec le programme pricipal suivant

```
int main(){
        vecteur a(5) ;
        vecteur b(2) ;
        b=a;
        return 0;
}
```

Opérateur =

- Rédefinir l'opérateur = pour pouvoir réaliser une affectation entre des objets déjà initialisés

- Pour une classe **NOMCLASSE** la méthode à ajouter pour rédefinir l'opérateur = :

  NOMCLASSE & operator=(NOMCLASSE&) ;

- Pour notre classe **vecteur** on va ajouter alors la méthode :

  vecteur & operator=(vecteur &);

Opérateur =

- Rédefinition de l'opérateur = pour notre classe vecteur :

```
vecteur & vecteur::operator=(vecteur & op){
    delete [] adr;
    adr = new double [nelem = op.nelem] ;
    for (int i=0; i<nelem; i++)   adr[i] = op.adr[i];
    cout << "+++ c'est une affectation +++ adr objet : " << this
    <<" +++ adr vecteur : " << adr <<endl;
    return *this;
}
```

Pour résumer

- Constructeur de recopie

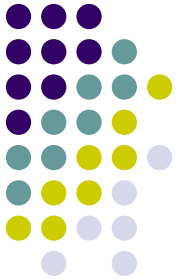
```
int main() {
    vecteur a(5);
    vecteur b=a;
    return 0;
}
```

```
void f(vecteur v){}
int main() {
    vecteur a(5)
    f(a) ;
    return 0;
}
```

- L'opérateur =

```
int main() {
    vecteur a(5);
    vecteur b(2);
    b=a ;
    return 0;
}
```

Forme canonique de Coplien



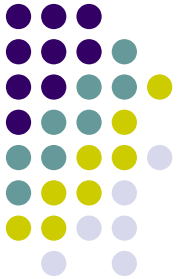
```
class T {
        public:
            T(); // Constructeur par défaut
            T(const T &); // Constructeur de recopie
            ~T (); // Destructeur éventuellement virtuel
            T &operator=(const T &);  // Operator d'affectation
};
```

Forme canonique de Coplien

- **Adapter et compléter votre projet pour pouvoir exécuter le programme suivant : combien d'objets vecteur vont être créés ?**

```
int main(){
        vecteur a(5);
        vecteur b=a;
        vecteur t[1];
        t[0]=a;
        return 0;
}
```

Objets membres



- Un membre d'un objet peut éventuellement être un objet.
- **class cercle {**

  Point centre;

  int rayon;

  cercle(int, int, int);

  };

  cercle::cercle(int abs, int ord, int ray) : centre(abs, ord) { … }
- Le constructeur de Point est appelé avant celui de cercle.
- Pour les destructeurs, c'est l'ordre inverse.
- On peut appeler une méthode de la classe Point :

  cercle a(1,2,3) ;

  a.centre.affiche() ;

Initialisation de membres dans l'en-tête d'un constructeur

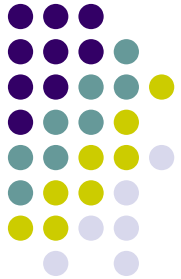
- La syntaxe que nous avons décrit pour transmettre des arguments à un constructeur d'un objet membre peut s'appliquer à n'importe quel membre, même s'il ne s'agit pas d'un objet.
- Par exemple:

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

- L'appel du constructeur point provoquera l'initialisation des membres x et y.
- Le constructeur est vide ici, puisque il n'y a rien de plus à faire pour remplacer notre constructeur classique:

```
Point(int abs=0,int ord=0){x=abs;y=ord;}
```

Initialisation de membres dans l'en-tête d'un constructeur



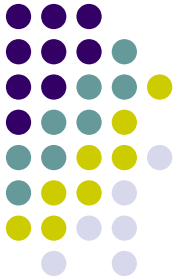
- Cette possibilité peut devenir indispensable Lorsque la donnée membre :

- Est qualifiée constante : cette donnée pourra recevoir une valeur initiale mais toute affectation est illicite.

- Est de type référence : dans ce cas l'affectation modifie la donnée référencée, l'initialisation est la seule possibilité pour installer la référence.

- Est une instance de classe dont le constructeur réclame des arguments(voir objets membres).

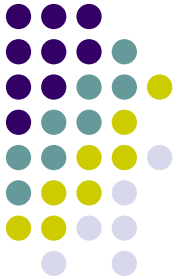
Initialisation de membres dans l'en-tête d'un constructeur



```
#ifndef POINT_H
#define POINT_H
class Point
{
    int x,y;
    public:
        Point(int,int);
        Point(Point &);
        ~Point();
        void afficher();
        void deplacer(int, int);
};
#endif // POINT_H
```

```
#ifndef CERCLE_H
#define CERCLE_H
#include "Point.h"
class Cercle
{
    Point centre;
    int rayon;
    public:
        Cercle(Point &,int );
        Cercle(int,int,int);
        ~Cercle();
        void deplacer(int,int);
        void afficher();
};
#endif // CERCLE_H
```

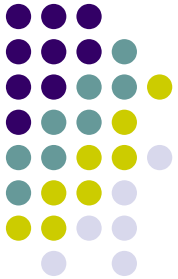
Initialisation de membres dans l'en-tête d'un constructeur



//Cercle.cpp

```
#include "Cercle.h"
#include "Point.h"
#include <iostream>
using namespace std;
Cercle::Cercle(Point & p0,int r):centre(p0),rayon(r){
        cout << "Constructeur de Cercle avec 2 args "<< endl;
}
Cercle::Cercle(int x0,int y0,int r):centre(x0,y0),rayon(r) {
        cout << "Constructeur de Cercle avec 3 args "<< endl;
}
Cercle::~Cercle() {}
void Cercle::deplacer(int dx, int dy){
    this->centre.deplacer(dx,dy);
}
void Cercle::afficher(){
    cout << "Le rayon = "<< rayon<< " et le centre est ";
    this->centre.afficher();
}
```

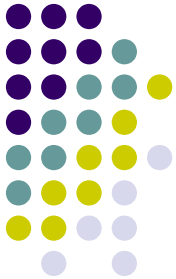
Initialisation de membres dans l'en-tête d'un constructeur



//Point.cpp

```
#include "Point.h"
#include <iostream>
using namespace std;
Point::Point(int x0,int y0):x(x0),y(y0){
    cout << "Constructeur de Point "<< endl;
}
Point::Point(Point & p1):x(p1.x),y(p1.y){
    cout << "Constructeur de recopie de Point "<< endl;
}
Point::~Point(){}
void Point::afficher(){
    cout << "(x = "<< x<< " et y = "<< y<<")"<< endl;
}
void Point::deplacer(int dx,int dy){
    x+=dx;
    y+=dy;
}
```

Initialisation de membres dans l'en-tête d'un constructeur



//main.cpp

```
#include <iostream>
#include "Point.h"
#include "Cercle.h"
using namespace std;

int main()
{
    Point a(2,2);
    Cercle cr(a,5);
    Cercle cr2(7,7,2);
    //cr=cr2;
    return 0;
}
```

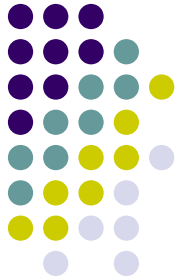
Héritage
(relations entre classes)

On peut noter trois sortes de relations entre deux classes A et B :

- Un objet A est composé d'un ou de plusieurs objets B, C, D
  - Relation d'inclusion (B,C et D sont des objets membres de A)
  - Exemple : Une voiture(A) a un moteur (B)

- Un objet A utilise B mais l'objet B ne fait pas partie de l'objet A et inversement. Relation de collaboration entre classes indépendantes (donnée membre de type pointeur pour matérialiser le lien)
  - Exemple : Une voiture(A) utilise une route(B)

- L' objet A est une sorte de l'objet B (Relation d'héritage)
  - Exemple : Une voiture (A) est un vehicule (B)



Héritage

- Donner à une classe toutes les caractéristiques d'une ou de plusieurs autres classes (appelées classes mères ou classes de base)

- La classe elle-même est appelée classe fille ou classe dérivée et héritera de tous les membres (données et fonctions) de ses classes de base « public ou protected »

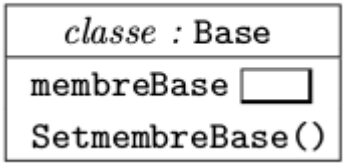
- L'intérêt majeur de l'héritage est de pouvoir définir de nouveaux attributs et de nouvelles méthodes pour la classe dérivée, qui viennent s'ajouter à ceux et celles héritées

- Par ce moyen on crée une hiérarchie de classes de plus en plus spécialisées

Héritage

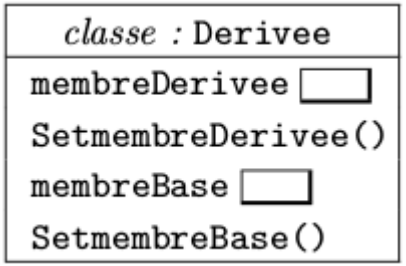
- Considérons par exemple la classe Base suivante :

```
class Base {
        public:
          int membreBase;
          void SetmembreBase(int valeurBase);
};
```



- Pour déclarer une classse dérivée de la classe Base :

```
class Derivee : public Base    // héritage public
{
        public:
          int membreDerivee;
          void SetmembreDerivee(int valeurDerivee);
};
```



Héritage

- De façon plus générale :

```
class A : public|protected|private B,
          public|protected|private C,
          public|protected|private D
{
      /*declarations de la classe A*/
};
```

Accès dans la classe de base	Type d'héritage		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	Non accessible	Non accessible	Non accessible

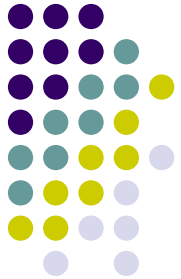
Héritage

- Constructeurs et héritage

```
class A{
   public :
     A(...) ;
     ...
} ;
```

```
class B  :  public A{
   public :
     B(...) ;
     ...
} ;
```

- Pour créer un objet de type B, il faut :
  - Créer un objet de type A (==> appel au constructeur de A)
  - Compléter par ce qui est spécifique à B (==> appel au constructeur de B)

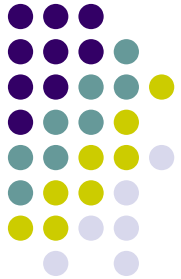


Héritage

- Constructeurs et héritage

  - Le programmeur spécifie l’appel au constructeur de la classe mère (à travers la liste d'initialisation) : un message d'erreur si aucun constructeur ne convient

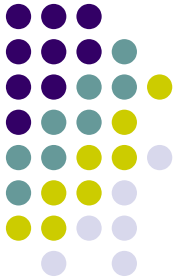
  - Pas d'appel specifié au niveau de la liste d'nitialisation
    - le constructeur par défaut "ou sans args" va être appelé : un message d'erreur s'il n'existe pas



Héritage

- Héritage et destructeur
  - Le destructeur de la classe mère est automatiquement appelé par le destructeur de la classe fille
  - => Le destructeur de la classe fille est t exécuté avant celui de la classe mère

Héritage

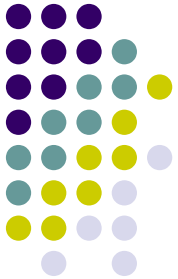


- Mise en oeuvre d'un exemple simple : gérer de points colorés

```
class PointCol : public Point
{
    public:
        PointCol(int,int,short);
        ~PointCol();
        PointCol(const PointCol& other);
        PointCol& operator=(const PointCol& other);

    private:
        short couleur;
};
```

```
class Point
{
    protected :
            int x,y;
    public:
        Point(int,int);
        Point(Point &);
        ~Point();
        void afficher();
        void deplacer(int, int);
};
#endif // POINT_H
```



Héritage

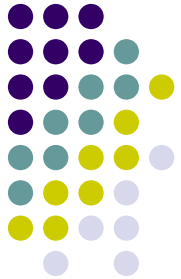
- Mise en oeuvre d'un exemple simple : gérér de points colorés

```
PointCol::PointCol(int x1,int y1,short c):Point(x1,y1),couleur(c)
{    cout << "Constructeur de PointCol "<< endl;}

PointCol::~PointCol(){cout << "Destructeur de PointCol "<< endl;}

PointCol::PointCol(const PointCol& other):Point(other),couleur(other.couleur)
{     cout << "Constructeur de recopie de PointCol "<< endl;}

PointCol& PointCol::operator=(const PointCol& rhs)
{
   if (this == &rhs) return *this; // handle self assignment
   this->x=rhs.x; this->y=rhs.y;
   this->couleur=rhs.couleur;
   cout << "C'est une affectation de PointCol"<< endl;
   return *this;
}
```



Héritage

- Mise en oeuvre d'un exemple simple : gérér de points colorés

```
Point::Point(int x0,int y0):x(x0),y(y0){ cout << "Constructeur de Point "<< endl;}

Point::Point(const Point & p1):x(p1.x),y(p1.y){
        cout << "Constructeur de recopie de Point "<< endl; }

Point::~Point(){    cout << "Destructeur de Point "<< endl;    }

void Point::afficher(){
        cout << "mes coordonnées sont (x = "<< x<< " et y = "<< y<<")"<< endl;
}
void Point::deplacer(int dx,int dy){
        x+=dx;
        y+=dy;
}
```

Héritage

- Utilisation des membres de la classe de base

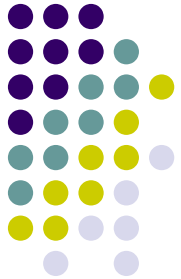
```
#include <iostream>
#include "PointCol.h"
using namespace std;

int main() {
    PointCol a(2,2,1);
    a.afficher();
    a.deplacer(5,5);
    a.afficher();
    return 0;
}
```

```
Constructeur de Point
Constructeur de PointCol
mes coordonnées sont (x = 2 et y = 2)
mes coordonnées sont (x = 7 et y = 7)
Destructeur de PointCol
Destructeur de Point

Process returned 0 (0x0)   execution time : 0.004 s
Press ENTER to continue.
```

Héritage



- Redéfinition / surcharge des fonctions membre

- Pour rédefinir la méthode afficher dans la classe PointCol, on ajoute :
  - la signature « void afficher() ; » dans le fichier PointCol.h
  - la définition suivante dans le fichier PointCol.cpp :

```
void PointCol::afficher(){
   cout << "PointCol : ma couleur = "<< couleur <<" et ";
   Point::afficher();
}
```

Héritage

- Après rédefinition de la méthode afficher dans PointCol

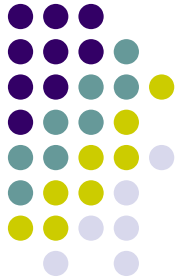
```
#include <iostream>
#include "PointCol.h"
using namespace std;

int main() {
    PointCol a(2,2,1);
    a.afficher();
    a.deplacer(5,5);
    a.afficher();
    return 0;
}
```

```
Constructeur de Point
Constructeur de PointCol
PointCol : ma couleur = 1 et mes coordonnées sont (x = 2 et y = 2)
PointCol : ma couleur = 1 et mes coordonnées sont (x = 7 et y = 7)
Destructeur de PointCol
Destructeur de Point

Process returned 0 (0x0)   execution time : 0,004 s
Press ENTER to continue.
```

Héritage



- Conversion d'un objet dérivé dans un objet de type de base

```
Point a(4,6);
PointCol b(3,5,2);
a=b;
a.afficher() ;
```

- Conversion de b dans le type Point et affectation du résultat à a par:
  - appel de l'opérateur d'affectation de la classe Point s'il a été surdéfini
  - par l'emploi de l'affectation par défaut sinon

Héritage

- Conversion d'un pointeur p sur une classe dérivée en un pointeur sur une classe de base

```
Point a(2,2);
PointCol b(4,4,1);
Point *p = &a;
PointCol *pCol=&b;
p=pCol;
p->afficher();
```

Typage statique : le compilateur appelle la méthode qui correspond au type défini pour la variable

- p pointe sur un objet de type PoinCol mais **p->afficher()** appelera la methode afficher() de la classe Point

Héritage

- Typage statique

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

int main() {
    Point a(2,2);
    PointCol b(4,4,1);
    Point *p = &a;
    PointCol *pCol=&b;
    p=pCol;
    p->afficher();
    return 0;
}
```

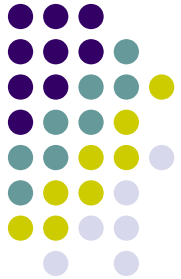
```
Constructeur de Point
Constructeur de Point
Constructeur de PointCol
mes coordonnées sont (x = 4 et y = 4)
Destructeur de PointCol
Destructeur de Point
Destructeur de Point

Process returned 0 (0x0)   execution time : 0,003 s
Press ENTER to continue.
```

Héritage

- Typage ou ligature dynamique (Polymorphisme)
  - À l'exécution quand on a un pointeur p qui pointe sur un objet de type (Point ou PointCol) **p->afficher()** invoquera afficher() de la classe réelle de l'objet pointé

```
Point a(4,6);
PointCol b(3,5,2);
Point *p = &a;
PointCol *pCol=&b
p=pCol;
p->afficher()
```



Héritage

- Pour utiliser le typage dynamique pour une méthode, dans la classe de base cette méthode doit être declarée de type virtual

```
class Point {
    protected:
        int x,y;
    public:
        Point(int,int);
        Point(const Point &);
        ~Point();
        virtual void afficher();
        void deplacer(int, int);
};
```

Héritage

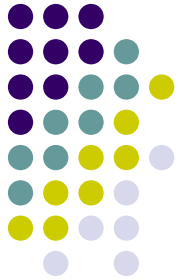
- Typage dynamique

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

int main() {
    Point a(2,2);
    PointCol b(4,4,1);
    Point *p = &a;
    PointCol *pCol=&b;
    p=pCol;
    p->afficher();
    return 0;
}
```

```
Constructeur de Point
Constructeur de Point
Constructeur de PointCol
PointCol : ma couleur = 1 et mes coordonnées sont (x = 4 et y = 4)
Destructeur de PointCol
Destructeur de Point
Destructeur de Point

Process returned 0 (0x0)   execution time : 0,004 s
Press ENTER to continue.
```

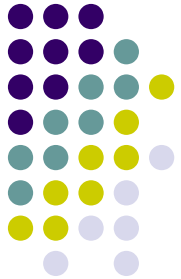


Héritage

- Polymorphisme : par défaut une méthode en C++ est non polymorphe

- Pour la rendre polymorphe, il faut :
  - déclarer la méthode virtuelle dans la classe mère
  - la redéfinir en respectant scrupuleusement la même signature dans la classe dérivée

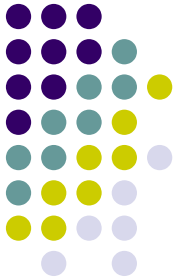
- C++ impose que toute classe possédant au moins une méthode virtuelle doit avoir un destructeur virtuel



Héritage

- Quand une méthode est déclarée virtuelle (mot clé virtual)
  - Cela précise au compilateur que lors des appels de la fonction on doit faire une ligature dynamique et non statique
  - Le choix de la méthode à exécuter est reporté au moment de l'exécution
  - Seule la méthode de la classe de base doit être déclarée virtuelle

Héritage



- Autre situation où la ligature dynamique est indispensable

```
#ifndef A_H
#define A_H
class A {
    public:
        A();
        ~A();
        void f();
        void g();
};
#endif // A_H
```

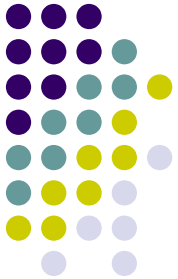
```
#ifndef B_H
#define B_H
#include <A.h>

class B : public A {
    public:
        B();
        ~B();
        void g();
};
#endif // B_H
```

```
A::A() { }
A::~A() { }
void A::f(){ g();}
void A::g(){cout << " g de A"<<endl;}
B::B() {}
B::~B(){}
void B::g(){cout << " g de B"<<endl;}

int main() {
    B b;
    b.f();
    return 0;
}
```

Héritage



- Autre situation où la ligature dynamique est indispensable

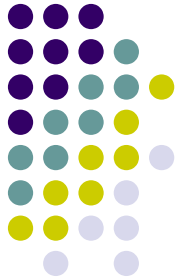
```
#ifndef A_H
#define A_H
class A {
    public:
        A();
        ~A();
        void f();
        virtuall void g();
};
#endif // A_H
```

```
#ifndef B_H
#define B_H
#include <A.h>

class B : public A {
    public:
        B();
        ~B();
        void g();
};
#endif // B_H
```

```
A::A() { }
A::~A() { }
void A::f(){ g();}
void A::g(){cout << " g de A"<<endl;}
B::B() {}
B::~B(){}
void B::g(){cout << " g de B"<<endl;}

int main() {
    B b;
    b.f();
    return 0;
}
```



Héritage

- Seule une fonction membre peut être virtuelle
- Redéfinition d'une fonction virtuelle n'est pas obligatoire
- Un constructeur ne peut pas être virtuel
- Un destructeur peut être virtuel
- Fonctions virtuelles pures ((définition nulle)) = outil pour créer une classe abstraite (une classe qu'on peut pas instancier)
  - **virtual void surface()=0;**
  - On ne sait pas comment implémenter cette méthode dans la classe mais on le saura dans les classes dérivées