Mecatronique – 2017/2018
Programmation C++
Leçon 7 : Les classes

Prof. *A. DARGHAM*

Sommaire

- Programmation procédurale et POO
- Introduction aux classes
- Création d'une instance d'une classe
- Constructeurs
- Paramètre implicite
- Fonctions membres privées
- Objets temporaires
- Données membres statiques
- Fonctions membres constantes

Sommaire

- Données membres constantes
- Fonctions amies
- Constructeur de recopie
- Destructeurs

Programmation procédurale & POO

- Programmation procédurale
  - Un programme procédural est formé d'une collection de données stockées dans des variables simples ou structurées, couplée d'un ensemble de fonctions réalisant certaines opérations sur ces données.
  - Les données et les fonctions qui les manipulent sont séparées les unes des autres.
  - Par exemple, dans un programme qui traite de la géométrie de rectangles, nous aurons besoin des données et des fonctions suivantes :

Programmation procédurale & POO

- **Données** :

```
double largeur; // stocke la largeur d'un rectangle */
double longueur;// stocke la longueur d'un rectangle
```

- **Fonctions** :

```
fixerDonnees(); // Range les valeurs dans largeur et
                // longueur d'un rectangle
afficherLargeur()
// affiche la largeur d'un rectangle
afficherLongueur()
// affiche la longueur d'un rectangle
afficherSurface()
// affiche la surface d'un rectangle
```

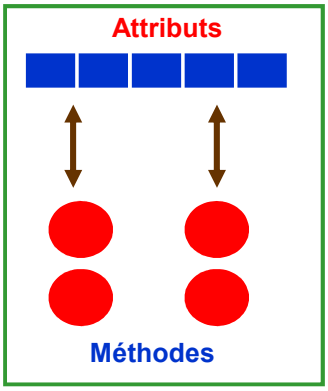
Programmation procédurale & POO

- Souvent, les **données** dans un **programme procédural sont passées** aux **fonctions** qui **réaliseront les opérations désirées**.
- La **programmation procédurale se concentre** alors **sur la création des fonctions** qui **réalisent des opérations les données du programme**.
- Lorsque **la structure d'une donnée a été modifiée**, **il faut absolument revoir le code qui manipule cette donnée** à fin d'accepter le nouveaux format.
- Cela **ajoute un travail additionnel aux programmeurs** pour maintenir le code, ce qui permet **d'exposer le code à des erreurs difficile à gérer**.

Programmation procédurale & POO

- **Programmation orientée objet (POO)**
  - La **programmation orientée objet se concentre sur la création des objets**.
  - Un **objet** est une **entité logique** qui **contient à la fois** les **données** et les **fonctions**.
  - Les **données d'un objet** sont appelées les **attributs** de cet objet.
  - Les **fonctions d'un objet** sont appelées les **méthodes de cet objet**.

Programmation procédurale & POO

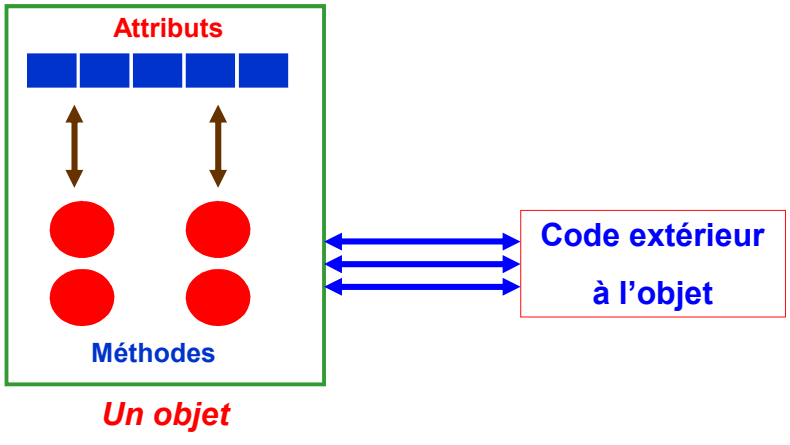


Un objet

Programmation procédurale & POO

- **Mécanismes de la POO**
  - **L'encapsulation** :
    - Par ce mécanisme, on peut **combiner les données et les fonctions** au sein d'une unique entité qui est **l'objet**.
  - **La protection des données** :
    - Par ce mécanisme, les **objets peuvent cacher les détails de leurs implémentations** au **monde extérieur**.
    - **Seules les fonctions membres d'un objet peuvent directement accéder et modifier** les **données membres de cet objet**.

Programmation procédurale & POO



Un objet

Programmation procédurale & POO

- Lorsqu'une **donnée membre d'un objet est cachée au code extérieur**, et que **l'accès à cette donnée est restreint aux fonctions membres de cet objet**, la **donnée sera protégée contre toute tentative d'altération**.
- De plus, **le code extérieur n'a pas besoin de connaître le format interne de l'objet**. Il a **uniquement besoin de connaître l'interface externe de l'objet et d'interagir avec l'objet à travers ses méthodes**.

Programmation procédurale & POO

- Si le **programmeur modifie la structure d'une donnée membre d'un objet**, **il ne modifiera que les fonctions membres** qui **manipulent cette donnée**.
- Cependant, **la manière avec laquelle le code extérieur interagit avec l'objet restera inchangée**.
- Un des **principaux objectifs** de la **POO** est de **laisser le code utilisant les données inchangé** lorsque la **représentation interne des données change**.

Programmation procédurale & POO

- Un **exemple de la vie courante** qui **illustre les techniques** de la **POO** est la **voiture**.
- Une **voiture**, **même si elle a une structure complexe**, **possède une interface très simple**.
- Si une **personne veux conduire une voiture**, **elle doit uniquement apprendre comment manipuler un nombre restreint d'organes de l'interface de la voiture** (**boîtier de démarrage**, **embrayage**, **accélérateur**, **freins** et **guidon**).

Programmation procédurale & POO

- Pour **démarrer le moteur de la voiture**, la **personne tourne simplement la clé de la voiture**.
- **Ce qui ce passe à l'intérieur de la voiture** est **inutile pour la personne**.
- Une voiture peut être donc conduite par des personnes **qui n'ont aucune connaissance de mécanique**, car les **voitures ont une interface très simple à comprendre et à manipuler**.
- Cela est bon pour les constructeurs d'automobiles, car cela signifie un nombre important de consommateurs.
- Cela est aussi bon pour les consommateurs, car il suffit juste d'apprendre un nombre très réduit de procédures pour conduire une voiture.

Introduction aux classes

- En C++, une **classe** est utilisée pour créer des **objets**.
- Une **classe sert de modèle** pour ses **objets**.
- Une **classe est une collection d'objets ayant** :
  - **La même structure** (les **données**)
  - **Les mêmes comportements** (les **fonctions**).
- En C++, les **attributs d'un objet** sont appelés des **données membres**, et les **méthodes d'un objet** sont appelées des **fonctions membres**.
- **Avant de créer un objet**, **il faut déclarer la classe à laquelle appartient cet objet**.

Introduction aux classes

- **Déclaration d'une classe C++**
  - En C++, une **classe** est un **type de données défini par le programmeur**.
  - Une **classe** C++ **consiste en un ensemble** de **données membres** (**variables**) et de **fonctions membres** (**fonctions**).

Introduction aux classes

- **Syntaxe de déclaration d'une classe en C++**

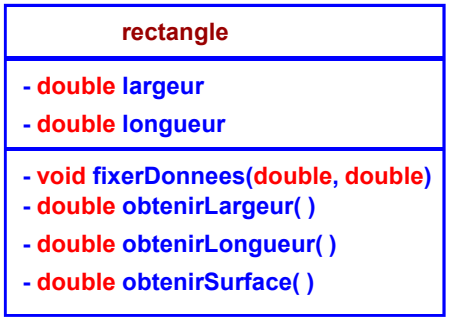
```
class Nom_de_la classe
{
    /* déclaration des données membres et  des
  fonctions membres de la classe */
};
```

Introduction aux classes

- **Exemple**

```
class rectangle
{
   // Données membres (Attributs)
   double largeur;
   double longueur;
   // Fonctions membres (Méthodes, ou Services)
   void   fixerDonnees(double, double);
   double obtenirLargeur();
   double obtenirLongueur();
   double obtenirSurface();
};
```

Introduction aux classes



Représentation graphique d'une classe par UML

Introduction aux classes

- **Qualificateurs d'accès (visibilité) des membres d'une classe C++ :**
  - En C++, une **donnée** ou une **fonction membre** peut être :
    - **Publique** : **accessible** par **n'importe quel code**.
    - **Privée** : **accessible uniquement à l'intérieur de la classe elle-même et ses amies.**
    - **Protégée : accessible uniquement à l'intérieur de la classe elle même**, **ses amies**, **ses classes dérivées** et **leurs amies**.

Introduction aux classes

- **Qualificateurs d'accès (visibilité) des membres d'une classe C++ :**
  - **Par défaut**, **tous les membres** d'une **classe C++** sont **privés**.
  - Pour **spécifier qu'un membre est publique**, on utilise le **qualificateur *public*** dans sa déclaration.
  - Pour **spécifier qu'un membre est privé**, on utilise le **qualificateur *private*** dans sa déclaration.
  - Pour **spécifier qu'un membre est protégé**, on utilise le **qualificateur *protected*** dans sa déclaration.

Introduction aux classes

- En général :
  - Les **données membres** sont **privées**.
  - Les **fonctions membres** sont **publiques**.
- En C++, une **classe** possède :
  - Une **interface** : c'est la **déclaration de la classe** (son ***nom***, ses ***données membres***, ses ***fonctions membres*** avec leurs ***visibilités*** [public | private | protected]).
  - Une **implémentation** : c'est la **définition de la classe** (les ***définitions de ses fonctions membres***).

Introduction aux classes

- Interface de la classe « rectangle » :

```
class rectangle
{
   private:
      double largeur;
      double longueur;
   public:
      void   fixerDonnees(double, double);
      double obtenirLargeur();
      double obtenirLongueur();
      double obtenirSurface();
};
```

Introduction aux classes

- Implémentation de la classe « rectangle » :

```
void rectangle::fixerDonnees(double l, double h)
{
   largeur = l;
   longueur = h;
}
double rectangle::obtenirLargeur()
{
   return largeur;
}
double rectangle::obtenirLongueur()
{
   return longueur;
}
```

Introduction aux classes

- **Implémentation de la classe « rectangle » :**

```
double rectangle::obtenirSurface()
{
    return largeur * longueur;
}
```

Introduction aux classes

- Dans l'**implémentation de la classe « rectangle »**, nous avons utilisé le **qualificateur** « **rectangle::** » comme **préfixe du nom de chaque fonction membre**.
- Cela est **nécessaire** pour la **définition de toute fonction membre écrite à l'extérieur de sa classe**.
- **L'opérateur de résolution de portée** « **::** » est utilisé pour **indiquer la classe de la fonction membre**, sinon le **compilateur ne pourra pas savoir à quelle classe appartient la fonction membre** (de plus, **deux classes différentes peuvent avoir la même fonction membre**).

Introduction aux classes

- Il est possible de définir une fonction membre à l'intérieur de sa **classe**.
- **Règle** :

  Toute fonction membre définie à l'intérieur de sa classe est considérée comme une fonction en ligne.

Introduction aux classes

- **Exemple :**

```
class rectangle
{
   private:
      double largeur;
      double longueur;
   public:
      void   fixerDonnees(double, double);
      double obtenirLargeur() { return largeur; }
      double obtenirLongueur(){ return longueur; }
      double obtenirSurface()
      { return largeur * longueur; }
};
```

Création d'une instance d'une classe

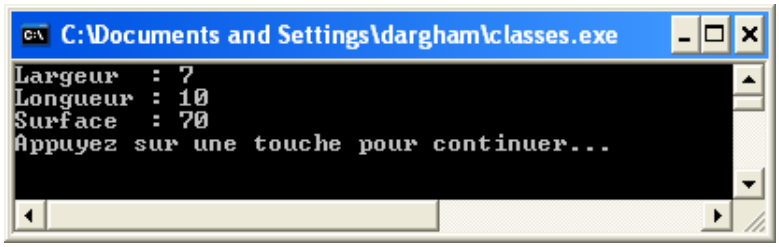
- **Objets et classes**
  - Une **classe** est un **type abstrait défini** par le programmeur.
  - Un **objet** est une **variable** de ce **type**.
  - En **POO**, on dit qu'**un objet est une instance d'une classe**.
  - **L'instanciation** est le **mécanisme** par lequel on **crée des objets** à **partir de leur classe**.

Création d'une instance d'une classe

- **Exemple**

```
int main()
{
   rectangle R; /* un objet de la classe rectangle */

   R.fixerDonnees(7.0, 10.0);
   cout << "Largeur  : " << R.obtenirLargeur() << endl;
   cout << "Longueur : " << R.obtenirLongueur() << endl;
   cout << "Surface  : " << R.obtenirSurface() << endl;
   system("pause");
   return 0;
}
```

Création d'une instance d'une classe



Création d'une instance d'une classe

- **Objets et classes**
  - Un **objet** (comme **R** dans l'exemple) **se déclare tout juste comme** une **variable ordinaire**.
  - Son **type** est le **nom de la classe** au **quelle elle appartient** (la classe « **rectangle** » dans notre exemple).
  - **C++** nous permet **d'étendre le langage** en nous offrant la possibilité de la **création de nouveaux types de données**.

Constructeurs

- **Initialisation des objets**
  - La **classe rectangle** de notre exemple utilise la **fonction membre** `fixerDonnees`() pour **initialiser ses objets**.
  - Il paraît naturel d'avoir cette **initialisation au moment de la déclaration des objets**, comme dans la déclaration des types simples :

```
int n = 22;
char * s = "Bonjour";
```

  - C++ permet **d'appliquer ce style d'initialisation simple et naturel** pour **les objets des classes possédant des fonctions spéciales** : les **constructeurs**.

Constructeurs

- **Constructeurs**
  - Un **constructeur** est une **fonction membre** qui est **automatiquement appelée lorsqu'un objet est déclaré**.
  - Un **constructeur** possède les **propriétés suivantes** :
    - **Il doit avoir** le **même nom que sa classe**.
    - **Il doit être déclaré sans type de retour**.
    - **Il peut avoir 0**, **1** ou **plusieurs arguments**.
    - **Il peut avoir un argument par défaut**.
    - **Il peut être surchargé**.

Constructeurs

- **Exemple**

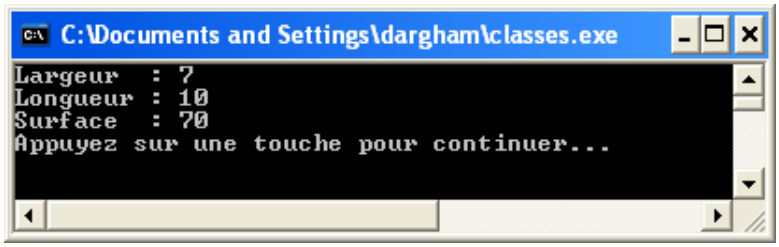
```
class rectangle
{
   private:
      double largeur;
      double longueur;
   public:
      rectangle(double l, double h) /* constructeur */
      { largeur = l; longueur = h; }
      double obtenirLargeur() { return largeur; }
      double obtenirLongueur(){ return longueur; }
      double obtenirSurface()
      { return largeur * longueur; }
};
```

Constructeurs

- **Exemple**

```
int main()
{
   rectangle R(7.0, 10.0); /* appel du constructeur */

   cout << "Largeur  : " << R.obtenirLargeur() << endl;
   cout << "Longueur : " << R.obtenirLongueur() << endl;
   cout << "Surface  : " << R.obtenirSurface() << endl;
   system("pause");
   return 0;
}
```

Constructeurs



Constructeurs

- **Exécution d'un constructeur**
  - Lorsque la déclaration de l'objet « `R` » s'exécute, le constructeur est automatiquement appelé et les valeurs « `7.0` » et « `10.0` » sont passées à ses paramètres « `l` » et « `h` ».
  - La fonction du constructeur affectera ces valeurs aux données membres de l'objet « `R` ».
  - La déclaration « `rectangle(7.0, 10.0);` » est équivalente au deux lignes :

```
rectangle R;
R.fixerDonnees(7.0, 10.0);
```

Constructeurs

- **Constructeur défini à l'extérieur de la classe**
  - On peut également **définir un constructeur à l'extérieur de sa classe**.
  - Comme exemple, voici la définition de notre constructeur « rectangle » à l'extérieur de sa classe rectangle :

```
rectangle::rectangle(double l, double h)
{
   largeur = l;
   longueur = h;
}
```

Constructeurs

- **Classe avec plusieurs constructeurs**
  - Une **classe peut avoir plusieurs constructeurs**.
  - Exactement comme n'importe quelle **fonction surchargée**, les **différents constructeurs** d'une **même classe seront distinguées par leurs listes d'arguments**.
  - Voici un exemple avec la classe « rectangle » :

Constructeurs

```
class rectangle
{
   private:
      double largeur;
      double longueur;
   public:
      rectangle();
      rectangle(double);
      rectangle(double, double);
      double obtenirLargeur() { return largeur; }
      double obtenirLongueur(){ return longueur; }
      double obtenirSurface()
      { return largeur * longueur; }
      void afficherInfos();
};
```

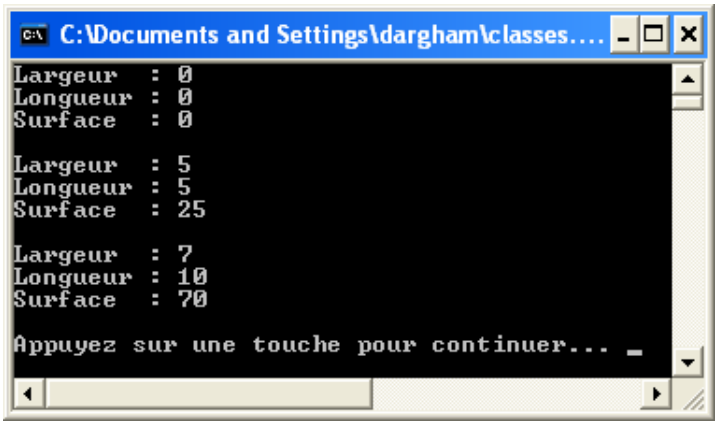
Constructeurs

```
rectangle::rectangle()
{
   largeur  = longueur = 0.0;
}
rectangle:: rectangle(double h)
{
   largeur  = longueur = h;
}
rectangle:: rectangle(double l, double h)
{
   largeur  = l;
   longueur = h;
}
```

Constructeurs

```
void rectangle::afficherInfos()
{
   cout << "Largeur  : " << obtenirLargeur() << endl;
   cout << "Longueur : " << obtenirLongueur() << endl;
   cout << "Surface  : " << obtenirSurface() << endl;
   cout << endl;
}
int main()
{
   rectangle R1, R2(5.0), R3(7.0, 10.0);
   R1.afficherInfos();
   R2.afficherInfos();
   R3.afficherInfos();
   system("pause");
   return 0;
}
```

Constructeurs



Constructeurs

- **Constructeur par défaut**
  - Parmi les différents constructeurs que peut avoir une classe quelconque, le **plus simple est celui n'ayant aucun argument**.
  - Ce constructeur est appelé le **constructeur par défaut de la classe**.
  - Si aucun **constructeur n'est déclaré dans la classe**, le compilateur va **créer automatiquement le constructeur par défaut pour cette classe**.
  - Mais, dès qu'un **constructeur est déclaré dans la classe**, **il n'y a plus de définition implicite** d'un **constructeur par défaut pour cette classe**.

Constructeurs

- **Exemple**

```
class A
{
   private:
      int a;
   public:
      void afficher() { cout << a << endl; }
};
int main()
{
   A x;
   x.afficher();  // constructeur par défaut
   system("pause");
   return 0;
}
```

Constructeurs

Constructeurs

- Le **constructeur par défaut initialise les données membres d'un objet par des valeurs aléatoires**.
- Il **peut être redéfinie** pour une **classe** (comme dans l'exemple de la classe « rectangle ») :

Constructeurs

```
class rectangle
{
   private:
      double largeur, longueur;
   public:
      rectangle();
      // redéfinition du constructeur par défaut
      rectangle(double);
     /* la suite de la déclaration ici */
};
```

Constructeurs

- **Constructeur avec liste d'initialisations**
  - La **majorité des constructeurs** ne font qu'une tâche **d'initialisation des données membres des objets**.
  - Pour cette raison, C++ fournit une **syntaxe très concise** qui permet de **simplifier l'écriture du code pour ces constructeurs**.
  - Cela est réalisé par une **liste d'initialisations**.

Constructeurs

- **Exemple**

```
class rectangle
{
   private:
      double largeur, longueur;
   public:
      rectangle(double l, double h) : largeur(x) ,
   longueur(h) { }
   // une liste d'initialisations

   /* la suite de la déclaration ici */
};
```

Constructeurs

- Les **instructions d'affectation** dans le **corps du constructeur ont été supprimées**.
- **Leurs actions** ont été **remplacées par la liste d'initialisations**.
- Notez que cette liste **débute par le symbole** « **:** » qui **précède le corps du constructeur**.
- Les **différentes valeurs d'initialisation** doivent être **séparées** par des « **,** ».

Constructeurs

- **Constructeur avec arguments par défaut**
  - Comme une **fonction ordinaire**, un **constructeur** peut avoir un ou plusieurs **arguments par défaut**.
  - Voici un exemple :

Constructeurs

```
class rectangle
{
   private:
      double largeur, longueur;
   public:
      rectangle(double l = 0.0, double h = 0.0)
      { largeur = l; longueur = h; }
      /* constructeur à arguments par défaut */
     // la suite de la déclaration
};
```

Constructeurs

- On peut même **combiner les deux dernières techniques** en utilisant une **liste d'initialisation** avec des **arguments par défaut**.
- Voici un exemple :

Constructeurs

```
class rectangle
{
   private:
      double largeur, longueur;
   public:
      rectangle(double l = 0.0, double h = 0.0)
      largeur(l), longueur(h) { }
      /* constructeur avec liste d'initialisation et à
          arguments par défaut */
};
int main()
{
   rectangle R1, R2(5.0), R3(7.0, 10.0);
   return 0;
}
```

Paramètre implicite

- Toute **fonction membre** d'une **classe** possède un **paramètre implicite**.
- Ce dernier est un **pointeur sur l'objet** qui **vient d'appeler la fonction membre**.
- Le **paramètre implicite** porte toujours le **nom** « **this** ».
- Par conséquent :
  - La valeur de « **this** » est **l'adresse de l'objet** qui a **appelé la fonction membre**.
  - La valeur de « ***this** » est **l'objet lui-même** qui a **appelé la fonction membre**.

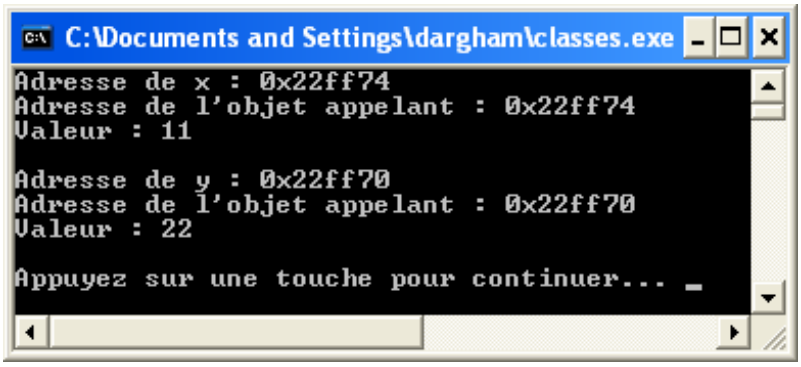
Paramètre implicite

```
class A
{
   private:
      int a;
   public :
      A(int n) : a(n) { }
      void print();
};
void A::print()
{
   cout << "Adresse de l'objet appelant : ";
   cout << this << endl;
   cout << "Valeur : " << this->a << endl;
   cout << endl;
}
```

Paramètre implicite

```
int main()
{
   A x(11), y(22);

   cout << "Adresse de x : " << &x << endl;
   x.print();

   cout << "Adresse de y : " << &y << endl;
   y.print();

   system("pause");
   return 0;
}
```

Paramètre implicite



Fonctions membres privées

- La **règle générale** de la **POO** exige que les **données membres soient privées** et que les **fonctions membres soient publiques**.
- Mais cette **règle n'est pas obligatoire**.
- Dans certains cas, il est utile de déclarer une ou plusieurs **fonctions membres** comme **privées**.
- Ces **fonctions membres ne peuvent être appelées qu'à l'intérieur de la classe elle-même**.
- Normalement, ces **fonctions membres rendent des services internes à la classe**.

Fonctions membres privées

```
class INT
{
   private:
      int a;
   public :
      INT(int n) : a(n) {}
      void afficher();
   private:
      bool estMultiple(int n);
      /* fonction membre privée */
};
```

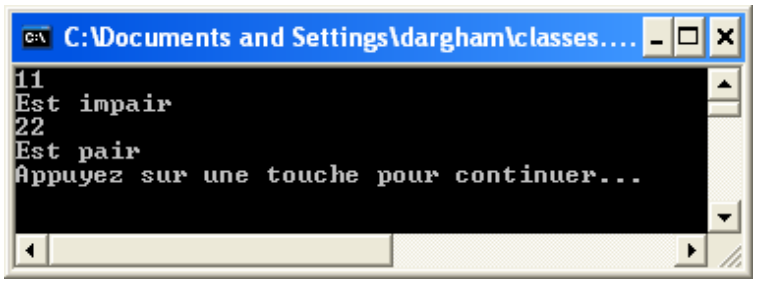
Fonctions membres privées

```
void INT::afficher()
{
   cout << a << endl;
   if(estMultiple(2))
     cout << "Est pair" << endl;
   else
     cout << "Est impair" << endl;
}

bool INT::estMultiple(int n)
{
   return a % n == 0;
}
```

Fonctions membres privées

```
int main()
{
   INT x(11), y(22);

   x.afficher();
   y.afficher();
   system("pause");
   return 0;
}
```

Fonctions membres privées



Objets temporaires

- Les **constructeurs** peuvent aussi créer des **objets temporaires**.
- Un **objet temporaire** est un **objet anonyme** (**sans nom**) qui est **crée en appelant un constructeur**, puis **automatiquement détruit** après **l'exécution d'une certaine fonction membre**.

Objets temporaires

- **Exemple**

```
int main()
{
    INT(11);
    /* un objet temporaire crée et détruit juste après
    l'exécution du constructeur */

    INT.print(77);
    /* un objet temporaire crée, puis appelle la fonction
    membre « print », et finalement détruit */

    system("pause");
    return 0;
}
```

Objets temporaires

Données membres statiques

- Une **donnée membre statique** est un **attribut dont la valeur est partagée** par **tous les objets de la classe**.
- En ce sens, une **donnée membre statique** est **rattachée à la classe** plutôt **qu'à l'objet**.
- Une **donnée membre statique** est en fait **une variable globale de la classe**.
- Une **donnée membre statique** sera utilisée pour **conserver une information sur l'ensemble des objets de la classe**.
- Une **donnée membre statique est déclarée en utilisant le mot-clé static**.

Données membres statiques

```
class INT
{
   private:
      int a;
      static int nbINT;
   // Une donnée membre statique
   public :
      INT(int n) : a(n) {}
      void afficher();
};
```

Données membres statiques

- À ce stade, la **donnée membre statique** est tout simplement déclarée, **mais non initialisée**.
- Une **donnée membre statique** doit être toujours **initialisée à l'extérieur de sa classe** :

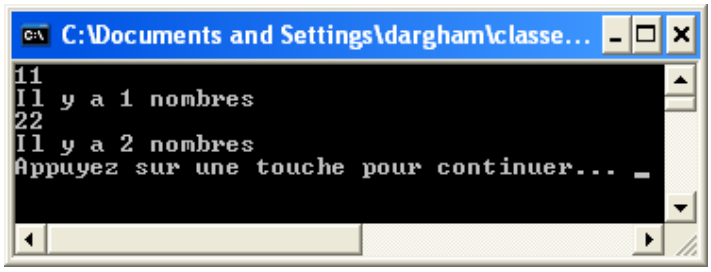
```
int INT::nbINT = 0;
// Initialisation de nbINT hors la classe
```

- Une **donnée membre statique** est ensuite **utilisée comme une donnée membre ordinaire**.

Données membres statiques

```
void INT::afficher()
{
   cout << a << endl;
   cout << "Il y a " << nbINT << " nombres" <<
  endl;
}
int main()
{
   INT x(11);
   x.afficher();
   INT y(22);
   y.afficher();
   system("pause");
   return 0;
}
```

Données membres statiques



Fonctions membres statiques

- Une **fonction membre statique** se **rattache à la classe** et **non pas aux objets de la classe**.
- Une **fonction membre statique** est **déclarée en utilisant le mot-clé static**.

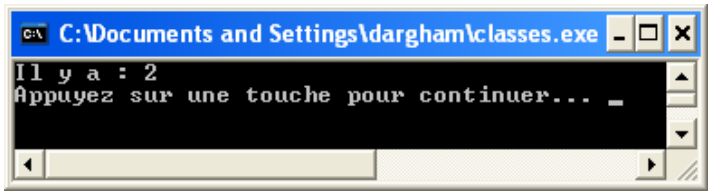
Fonctions membres statiques

```
class INT
{
   private:
      int a;
      static int nbINT;
   // Une donnée membre statique
   public :
      INT(int n) : a(n) {}
      void afficher();
      static int get_nbINT();
   // Une fonction membre statique
};
```

Fonctions membres statiques

```
int INT::get_nbINT()
{
   return nbINT;
}
int main()
{
   INT x(11), y(22);

   cout << "Il y a : " << INT::get_nbINT() << endl;
   system("pause");
   return 0;
}
```

Fonctions membres statiques



Fonctions membres constantes

- Une **fonction membre** peut être qualifiée **constante** par l'ajout du **mot-clé const** à **droite de son prototype**.

Fonctions membres constantes

```
class Point
{
   private:
      int abs, ord;
   public:
      Point(int x = 0, int y = 0) : abs(x), ord(y) {}
      void afficher() const;
      // une fonction membre constante
      int obtenirAbs() const { return abs; }
      // une autre fonction membre constante
};
```

Fonctions membres constantes

```
void Point::afficher() const
{
   cout << "Je suis en (" << abs;
   cout << ", " << ord << ")" << endl;
}
int main()
{
   Point A(-3, 7);

   A.afficher();
   cout << "Mon abscisse est :  " << A.obtenirAbs();
   cout << endl;
   system("pause");
   return 0;
};
```

Fonctions membres constantes

Fonctions membres constantes

- Une **fonction membre constante ne peut modifier aucun attribut** de **l'objet sur lequel elle s'applique** :

```
int Point::obtenirAbs() const
{
   abs++;
   /* Erreur : tentative de modification d'un attribut de
      l'objet appelant */
   return abs;
}
```

Fonctions membres constantes

- Un **objet constant** est un objet dont **on interdit la modification de toutes ses données membres**.
- Un **objet non constant** est un objet dont les **données membres peuvent changer** au cours de l'exécution du programme.
- Un **objet constant** se déclare en utilisant le mot-clé « **const** » :

```
Point A(-3, 7);
// un objet « Point » non constant
const Point B(3, 0);
// un objet « Point » constant
```

Fonctions membres constantes

- Une **fonction membre constante s'applique bien sur des objets constants ou non**.
- Cependant, une **fonction membre non constante ne s'applique que sur des objets non constants**.
- **Ne pas déclarer constante une fonction membre qui est effectivement constante** a une **conséquence grave** : **cela interdit son application sur des objets constants**.

Fonctions membres constantes

```
class Point
{
   private:
      int abs, ord;
   public:
      Point(int x = 0, int y = 0) : abs(x), ord(y) {}
      void afficher(); // non constante
      int obtenirAbs() const { return abs; } // constante
};
void Point::afficher()
{
   cout << "Je suis en (" << abs;
   cout << ", " << ord << ")" << endl;
}
```

Fonctions membres constantes

```
int main()
{
   Point A(-3, 7);       // un objet non constant
   const Point B(3, 2); // un objet constant

   cout << A.obtenirAbs() << endl;  // valide
   cout << B.obtenirAbs() << endl;  // valide
   A.afficher();  // valide
   B.afficher();  // non valide
   system("pause");
   return 0;
};
```

Données membres constantes

- Un **objet** peut avoir un **attribut constant**.
- Imaginons, un **point** ayant une **couleur fixe** et se **déplaçant** dans le plan :

```
class Point
{
   private:
      int abs, ord;
      const int coul; // une donnée membre constante
   public:
      Point(int x = 0, int y = 0, int c = 0);
      void afficher() const;
};
```

Données membres constantes

```
Point::Point(int x, int y, int c):coul(c)
{
   abs = x;
   ord = y;
}
void Point::afficher() const
{
   cout << "A cet instant, je suis en (" << abs;
   cout << ", " << ord << ")" << endl;
   cout << "Mais, je suis toujours de couleur ";
   cout << coul << endl;
}
```

Données membres constantes

```
int main()
{
   Point A(-3, 7, 9);

   A.afficher();
   system("pause");
   return 0;
}
```

Données membres constantes

Données membres constantes

- Attention au constructeur **Point** :

```
Point::Point(int x, int y, int c):coul(c)
{
   abs = x;
   ord = y;
}
```

- Il ne peut être écrit sous la forme suivante :

```
Point::Point(int x = 0, int y = 0, int c)
{
   abs = x;
   ord = y;
   coul = c;  // Erreur
}
```

Données membres constantes

- Dans cet exemple, il serait **impossible de déclarer une fonction membre** qui **tente de modifier une donnée membre constante** telle que « couleur ».
- Mais il est **possible de définir d'autres fonctions membres** qui **modifient les données membres non constantes**.

Données membres constantes

```
class Point
{
   private:
      int abs, ord;
      const int coul; // une donnée membre constante
   public:
      Point(int x = 0, int y = 0, int c = 0);
      void afficher() const;
      void fixerAbs(int x) { abs = x; } // Possible
      void incrementerOrd() { ord++; }  // Possible
      void changerCouleur(int c)
      { coul = c; } // Erreur : c'est impossible
};
```

Fonctions amies

- **Exemple 1**
  - On veut définir une **fonction usuelle** « **`estAvant`** » qui teste si un point A est avant un autre point B :

```
bool estAvant(Point A, Point B)
{
   return (A.abs < B.abs);
   // Erreur : A.abs et B.abs sont privées
}
```

  - La **fonction usuelle** « **`estAvant`** » **n'a pas le droit d'accéder aux données membres privées** de ses **deux arguments de type** « Point ».

Fonctions amies

- Une **solution à ce problème consiste** à **déclarer cette fonction** comme **amie de la classe** « Point » :

```
class Point
{
   private:
      int abs, abs;
   public:
      // fonctions membres de la classe
      friend bool estAvant(Point, Point);
};
```

Fonctions amies

- La fonction « `estAvant` » **restera une fonction usuelle**, mais elle **a le privilège d'accéder aux données membres de la classe** « Point ».

Fonctions amies

- **Exemple 2**
  - On veut définir une **fonction usuelle** « `produit` » qui multiplie un point A par un entier n :

```
Point A(1, 3);
Entier n(2);
Point B = produit(A, n);
B.afficher(); // affichera (2, 6)
```

  - La **fonction usuelle** « `produit` » **doit accéder aux données membres privées** des **deux classes** « Point » et « Entier », **ce qui est impossible**.

Fonctions amies

- Une **solution consiste à déclarer** cette **fonction comme amie des deux classes** :

```
class Point;
class Entier;
class Point
{
   private:
      int abs, abs;
   public:
      // fonctions membres de la classe Point
      friend Point produit(Point, Entier);
};
```

Fonctions amies

```
class Entier
{
   private:
      int val;
   public:
      // fonctions membres de la classe Entier
      friend Point produit(Point, Entier);
};
```

Fonctions amies

```
Point produit(Point A, Entier n)
{
   Point P;

   P.abs = n.val * A.abs;
   P.ord = n.val * A.ord;
   return P;
};
```

Fonctions amies

- **Exemple 3**
  - Il arrive que toutes les fonctions membres d'une classe soient amies d'une autre classe :

```
class A
{
   // . . . . . .
   friend class B;
};
```

  - On dit que la **classe** « B » **est amie** de la **classe** « A ».

Fonctions amies

- **Exemple 3**
  - **Toutes les fonctions membres de la classe** « B » **peuvent accéder à toutes les données membres privées de la classe** « A ».

Constructeur de recopie

- **Définition**
  - Le **constructeur de recopie** est un **constructeur spécial** qui est **appelé lorsqu'un objet est crée** et **initialisé par les données d'un autre objet** déjà crée.
  - Son **rôle** est d'**initialiser un objet** par **recopie des valeurs des données membres** d'un **autre objet**.
  - Sa **syntaxe de déclaration** pour une **classe nommée** « `x` » est :

```
x(const x &);
```

Constructeur de recopie

- **Constructeur de recopie implicite**
  - Si le programmeur **n'a pas défini explicitement** un **constructeur de recopie** pour une **classe**, le **compilateur va implicitement lui créer un**.
  - Ce dernier effectue une **initialisation** par **recopie des données membre à membre**.
  - Ce **comportement implicite** est **suffisant dans un grand nombre de cas**.
  - Mais, **dans d'autre cas** (par exemple si les **objets sont de taille variable**), la **définition explicite** d'un **constructeur de recopie s'impose**.

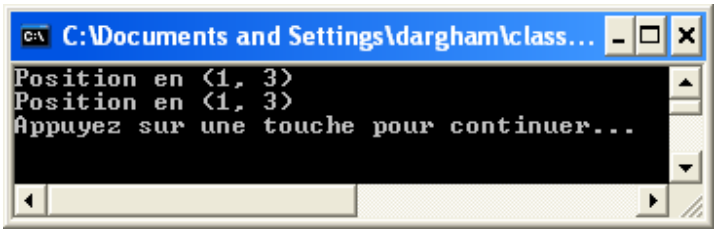
Constructeur de recopie

```
#include <iostream>
using namespace std;
class Point
{
   private:
      int abs, ord;
   public:
      Point(int x = 0, int y = 0) : abs(x), ord(y) {}
      void afficher() const;
};

void Point::afficher() const
{
   cout << "Position en (" << abs;
   cout << ", " << ord << ")" << endl;
}
```

Constructeur de recopie

```
int main()
{
   Point A(1, 3);
   Point B(A);
   /* appel du constructeur par recopie implicite */

   B.afficher();

   Point C = A;
   /* appel du constructeur par recopie implicite */

   C.afficher();

   system("pause");
   return 0;
}
```

Constructeur de recopie



Constructeur de recopie

```
#include <iostream>
using namespace std;
class Point
{
   private:
      int abs, ord;
   public:
      Point(int x = 0, int y = 0) : abs(x), ord(y) {}
      Point(const Point &);
      /* constructeur de recopie explicite */
      void afficher() const;
};
```

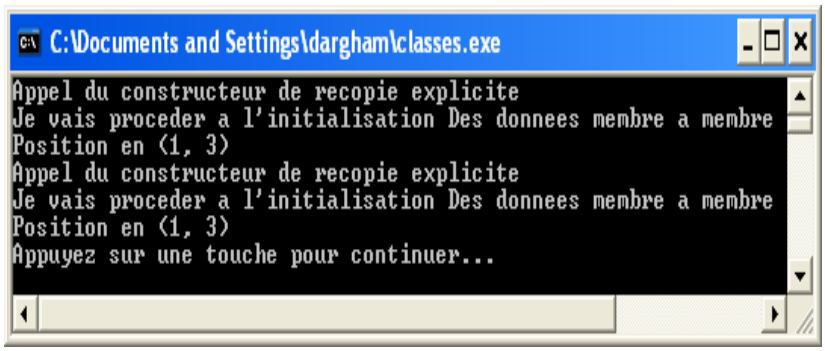
Constructeur de recopie

```
void Point::afficher() const
{
   cout << "Position en (" << abs;
   cout << ", " << ord << ")" << endl;
}
Point::Point(const Point & P)
{
   cout << "Appel du constructeur de recopie explicite";
   cout << "\nJe vais proceder a l'initialisation ";
   cout << "Des donnees membre a membre" << endl;
   abs = P.abs;
   ord = P.ord;
}
```

Constructeur de recopie

```
int main()
{
   Point A(1, 3);
   Point B(A);
   /* appel du constructeur par recopie explicite */

   B.afficher();

   Point C = A;
   /* appel du constructeur par recopie explicite */

   C.afficher();

   system("pause");
   return 0;
}
```

Constructeur de recopie



Destructeurs

- **Définition**
  - Le **destructeur** est une **fonction membre spéciale** dont le **rôle** est de **libérer l'espace mémoire occupée par un objet**.
  - Sa **syntaxe de déclaration** pour une **classe nommée** « `x` » est :

    `~x();`
  - Une **classe ne peut posséder qu'un seul destructeur** qui est **toujours sans arguments**.

Destructeurs

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

class Point
{
   private:
      int abs, ord;
      static int nbPoints;
   public:
      Point(int x = 0, int y = 0);
      ~Point(); /* destructeur */
      void afficher() const;
};
```

Destructeurs

```
int Point::nbPoints = 0;
Point::Point(int x, int y)
{
   abs = x;
   ord = y;
   cout << "Appel du constructeur" << endl;
   nbPoints++;
   cout << "Il y a " << nbPoints << " points" << endl;
}
Point::~Point()
{
   cout << "Appel du destructeur" << endl;
   nbPoints--;
   cout << "Il reste " << nbPoints << " points" << endl;
}
```

Destructeurs

```
void Point::afficher() const
{
   cout << "Position en (" << abs;
   cout << ", " << ord << ")" << endl;
}
int main()
{
   Point A(1, 2);
   A.afficher();
   {  /* un bloc interne */
      Point B(3, 4);
      B.afficher();
   }
   system("pause");
   return 0;
}
```

Destructeurs