# Chapitre 4 : Classes

## I52 Programmation Orienté 0bjet — C++

Valérie Gillot

Septembre 2019

# Plan du chapitre

Section 1

# Objet, classe et membre

- Déclaration et définition

# Classe

## Une classe

est un type abstrait de données, organisée en champs qui regroupe des objets ayant les mêmes propriétés :

- une zone mémoire allouée
- des fonctions qui manipulent ou modifient la zone mémoire décrivant l'objet

En C++, c'est le type class

# Déclaration

## La déclaration

- est la description du type de l'objet avec la liste de ces champs
- se fait dans un fichier en-tête d'extension `.h`

## Les champs

Il y a deux types de champs :

- les attributs (ou données membres) qui correspondent à la zone mémoire allouée
- les méthodes (ou fonctions membres) qui sont les fonctions manipulent ou modifient la zone mémoire

# Déclaration

## Syntaxe de la déclaration

```
class Nom {
private : //
     optionnel
...
public :
...
};
```

## Exemple

```
class Point{
private :
// attributs ou donnees membres
  int x,y;
public :
// methodes ou fonctions membres
  void Affiche();
  void Initialise(int, int);
};
```

## Mise en garde

Attention de ne pas oublier le ; à la fin de la déclaration

# Déclaration

## Syntaxe de la déclaration

```
class Nom {
private : //
    optionnel
...
public :
...
};
```

## Exemple

```
class Point{
private :
// attributs ou donnees membres
  int x,y;
public :
// methodes ou fonctions membres
  void Affiche();
  void Initialise(int, int);
};
```

## Mise en garde

Attention de ne pas oublier le ; à la fin de la déclaration

# Déclaration

## Syntaxe de la déclaration

```
class Nom {
private : //
    optionnel
...
public :
...
};
```

## Exemple

```
class Point{
private :
// attributs ou donnees membres
  int x,y;
public :
// methodes ou fonctions membres
  void Affiche();
  void Initialise(int, int);
};
```

## Mise en garde

Attention de ne pas oublier le ; à la fin de la déclaration

# Définition

## La définition d'une classe

- est la définition des méthodes qui manipulent l'objet déclarées dans le `.h`
- se fait dans un fichier d'extension `.cc`

## Opérateur de portée ::

Une méthode est définie en utilisant le nom de la classe et de l'opérateur de portée `NomDeClasse :: NomDeMethode`

## Exemple de définition

```
void Point::Initialise(int a, int b)
{
x=a;
y=b;
}
```

# Définition

## La définition d'une classe

- est la définition des méthodes qui manipulent l'objet déclarées dans le `.h`
- se fait dans un fichier d'extension `.cc`

## Opérateur de portée ::

Une méthode est définie en utilisant le nom de la classe et de l'opérateur de portée `NomDeClasse :: NomDeMethode`

## Exemple de définition

```
void Point::Initialise(int a, int b)
{
x=a;
y=b;
}
```

# Définition

## La définition d'une classe

- est la définition des méthodes qui manipulent l'objet déclarées dans le `.h`
- se fait dans un fichier d'extension `.cc`

## Opérateur de portée ::

Une méthode est définie en utilisant le nom de la classe et de l'opérateur de portée `NomDeClasse :: NomDeMethode`

## Exemple de définition

```
void Point::Initialise(int a, int b)
{
x=a;
y=b;
}
```

# Définition

## La définition dans la déclaration

On peut définir une méthode dans la déclaration de la classe sans le prefixe `inline`

### Exemple de définition

```
class Point{
private :
// attributs ou donnees membres
  int x,y;
public :
// methodes ou fonctions membres
  void Affiche();
  void Initialise(int a, int b){x=a;x=b;}
  };
```

# Définition

## La définition dans la déclaration

On peut définir une méthode dans la déclaration de la classe sans le prefixe `inline`

### Exemple de définition

```
class Point{
private :
// attributs ou donnees membres
  int x,y;
public :
// methodes ou fonctions membres
  void Affiche();
  void Initialise(int a, int b){x=a;x=b;}
  };
```

# Définition

## La définition dans la déclaration

On peut définir une méthode dans la déclaration de la classe sans le prefixe `inline`

### Exemple de définition

```
class Point{
private :
// attributs ou donnees membres
  int x,y;
public :
// methodes ou fonctions membres
  void Affiche();
  void Initialise(int a, int b){x=a;x=b;}
  };
```

Section 2

# Instance de classe, attribut et méthode

# Instance de classe : objet

## Classe et objet

Une classe est la description générale d'une famille d'objets et un objet est une instance de classe.

## Attribut

Un attribut est une variable d'instance d'une classe (donnée membre)

## Méthode

Une méthode est une fonction membre de la classe.

# Instance de classe : objet

## Classe et objet

Une classe est la description générale d'une famille d'objets et un objet est une instance de classe.

## Attribut

Un attribut est une variable d'instance d'une classe (donnée membre)

## Méthode

Une méthode est une fonction membre de la classe.

# Instance de classe : objet

## Classe et objet

Une classe est la description générale d'une famille d'objets et un objet est une instance de classe.

## Attribut

Un attribut est une variable d'instance d'une classe (donnée membre)

## Méthode

Une méthode est une fonction membre de la classe.

# Accès aux membres d'une classe

## Accès aux champs d'un objet

- L'opérateur `.` permet d'accéder aux membres (attributs ou méthodes) d'un objet
- l'opérateur `->` permet d'accéder aux membres d'un pointeur sur un objet.

## Accès aux membres

```
int main()
{
Point P;
Point * ptP;
ptP= new Point;
P.Initialise(2,3);
ptP->Initialise(4,5);
return 0;
}
```

- `P` est une instance de la classe `Point`
- `ptP` est l'adresse d'une instance de la classe `Point`
- à la place de `ptP->Initialise(4,5)`, on aurait pu écrire `(*ptP).Initialise(4,5)`

# Accès aux membres d'une classe

## Accès aux champs d'un objet

- L'opérateur `.` permet d'accéder aux membres (attributs ou méthodes) d'un objet
- l'opérateur `->` permet d'accéder aux membres d'un pointeur sur un objet.

## Accès aux membres

```
int main()
{
Point P;
Point * ptP;
ptP= new Point;
P.Initialise(2,3);
ptP->Initialise(4,5);
return 0;
}
```

- `P` est une instance de la classe `Point`
- `ptP` est l'adresse d'une instance de la classe `Point`
- à la place de `ptP->Initialise(4,5)`, on aurait pu écrire `(*ptP).Initialise(4,5)`

# Accès aux membres d'une classe

## Accès aux champs d'un objet

- L'opérateur `.` permet d'accéder aux membres (attributs ou méthodes) d'un objet
- l'opérateur `->` permet d'accéder aux membres d'un pointeur sur un objet.

## Accès aux membres

```
int main()
{
Point P;
Point * ptP;
ptP= new Point;
P.Initialise(2,3);
ptP->Initialise(4,5);
return 0;
}
```

- `P` est une instance de la classe `Point`
- `ptP` est l'adresse d'une instance de la classe `Point`
- à la place de `ptP->Initialise(4,5)`, on aurait pu écrire `(*ptP).Initialise(4,5)`

# Méthodes constantes

## Syntaxe de la déclaration

```
type identMethode(parametres) const;
```

## Sémantique

Les méthodes constantes ne modifient pas les attributs de la classe

## Exemple

```
class Point{
private :
  int x,y;// attributs ou donnees membres
public :
// methodes
  void Affiche() const;// methode constante
  void Initialise(int a, int b){x=a;x=b;}
  };
```

# Méthodes constantes

## Syntaxe de la déclaration

```
type identMethode(parametres) const;
```

## Sémantique

Les méthodes constantes ne modifient pas les attributs de la classe

## Exemple

```
class Point{
private :
  int x,y;// attributs ou donnees membres
public :
// methodes
  void Affiche() const;// methode constante
  void Initialise(int a, int b){x=a;x=b;}
  };
```

# Méthodes constantes

## Syntaxe de la déclaration

```
type identMethode(parametres) const;
```

## Sémantique

Les méthodes constantes ne modifient pas les attributs de la classe

## Exemple

```
class Point{
private :
  int x,y;// attributs ou donnees membres
public :
// methodes
  void Affiche() const;// methode constante
  void Initialise(int a, int b){x=a;x=b;}
  };
```

# Encapsulation et accessibilité

## Encapsulation

Les membres (attributs ou méthodes) d'une classe peuvent être :

- publics (qualificatif `public`)
- privés (qualificatif `private`)
- protégés (qualificatif `protected`)

## Les membres privés

Les membres déclarés `private` sont

- accessibles pour les fonctions membres (méthodes) ou les fonctions amies de la classe.
- inaccessibles partout ailleurs

## `public`

Les membres déclarés `public` sont accessibles depuis n'importe quelle fonction.

# Encapsulation et accessibilité

## Encapsulation

Les membres (attributs ou méthodes) d'une classe peuvent être :

- publics (qualificatif `public`)
- privés (qualificatif `private`)
- protégés (qualificatif `protected`)

## Les membres privés

Les membres déclarés `private` sont

- accessibles pour les fonctions membres (méthodes) ou les fonctions amies de la classe.
- inaccessibles partout ailleurs

## `public`

Les membres déclarés `public` sont accessibles depuis n'importe quelle fonction.

# Encapsulation et accessibilité

## Encapsulation

Les membres (attributs ou méthodes) d'une classe peuvent être :

- publics (qualificatif `public`)
- privés (qualificatif `private`)
- protégés (qualificatif `protected`)

## Les membres privés

Les membres déclarés `private` sont

- accessibles pour les fonctions membres (méthodes) ou les fonctions amies de la classe.
- inaccessibles partout ailleurs

## `public`

Les membres déclarés `public` sont accessibles depuis n'importe quelle fonction.

# Encapsulation et accessibilité

## Les rôles du programmeur

Le concepteur de la classe (C) et l'utilisateur de la classe (U)

- les membres privés sont accessibles pour (C), inaccessible pour (U)
- les membres publics sont accessibles pour (C) et (U)

# Accessibilité aux membres d'une classe

## point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
// methodes
  void Affiche();
  void Initialise(int,int);
  Point Somme(Point);
  };
```

## point.cc (C)

```
Point Point::Somme(Point P)
// retourne le point somme de l'
    objet courant et du point P
{
Point S;
S.x = x + P.x;
S.y = y + P.y;
return S;
}
```

# Accessibilité aux membres d'une classe

## point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
// methodes
  void Affiche();
  void Initialise(int,int);
  Point Somme(Point);
  };
```

## point.cc (C)

```
Point Point::Somme(Point P)
// retourne le point somme de l'
    objet courant et du point P
{
Point S;
S.x = x + P.x;
S.y = y + P.y;
return S;
}
```

# Accessibilité aux membres d'une classe

## main.cc (U)

```
int main ()
{
Point P, Q, R;
P.Initialise(2,3); // OK
Q.x=3; // KO
Q.y=4; //KO
Q.Initialise(3,4); // OK
R=P.Somme(Q); // OK
cout<<R.x<<R.y<<endl; // KO
return 0;
}
```

## Exercice

Écrire la méthode `Affiche` pour la classe `Point`.

# Accessibilité aux membres d'une classe

## main.cc (U)

```
int main ()
{
Point P, Q, R;
P.Initialise(2,3); // OK
Q.x=3; // KO
Q.y=4; //KO
Q.Initialise(3,4); // OK
R=P.Somme(Q); // OK
cout<<R.x<<R.y<<endl; // KO
return 0;
}
```

## Exercice

Écrire la méthode `Affiche` pour la classe `Point`.

# Encapsulation

## Remarques

1. Le type `struct` est une classe dont tous les champs sont publics par défaut, on peut aussi utiliser `private` ou `public`
2. En général les attributs d'une classe sont privés pour masquer le fonctionnement interne et protéger l'objet
3. En C++, contrairement aux autres langages de POO, l'encapsulation interdit au lieu de cacher.

# Encapsulation

## Remarques

1. Le type `struct` est une classe dont tous les champs sont publics par défaut, on peut aussi utiliser `private` ou `public`
2. En général les attributs d'une classe sont privés pour masquer le fonctionnement interne et protéger l'objet
3. En C++, contrairement aux autres langages de POO, l'encapsulation interdit au lieu de cacher.

# Encapsulation

## Remarques

1. Le type `struct` est une classe dont tous les champs sont publics par défaut, on peut aussi utiliser `private` ou `public`
2. En général les attributs d'une classe sont privés pour masquer le fonctionnement interne et protéger l'objet
3. En C++, contrairement aux autres langages de POO, l'encapsulation interdit au lieu de cacher.

Section 3

# Constructeur

# Constructeurs

## Constructeur

Un constructeur est une méthode particulière de la classe

- elle porte le même nom que la classe
- elle ne retourne rien même pas `void`
- ne contient pas d'instruction `return`

## Rôle du constructeur

1. Allouer l'espace mémoire pour les attributs nécessitant une allocation dynamique
2. Initialiser les attributs

# Constructeurs

## Constructeur

Un constructeur est une méthode particulière de la classe

- elle porte le même nom que la classe
- elle ne retourne rien même pas `void`
- ne contient pas d'instruction `return`

## Rôle du constructeur

1. Allouer l'espace mémoire pour les attributs nécessitant une allocation dynamique
2. Initialiser les attributs

# Constructeurs

## Création d'un objet

- les constructeurs sont automatiquement appelés par le compilateur lors de la création d'un objet de la classe (définition + appel du constructeur)
- l'appel à un constructeur
  - se fait lors de la déclaration d'un objet
  - se fait implicitement ou explicitement selon le cas

# Sur-définition des constructeurs

## Sur-définition

Il peut y avoir plusieurs constructeurs (sur-définition), ils doivent se distinguer par leur signature. Les plus courant :

- un constructeur par défaut (sans paramètre)
- plusieurs constructeurs avec paramètres
- un constructeur par copie (clonage)

## Constructeur par défaut

Deux possibilités :

- soit sans paramètre
- soit tous ses paramètres sont initialisés par défaut

## Constructeur par copie

Ce constructeur permet de cloner un objet transmis en paramètre par une référence constante.

# Sur-définition des constructeurs

## Sur-définition

Il peut y avoir plusieurs constructeurs (sur-définition), ils doivent se distinguer par leur signature. Les plus courant :

- un constructeur par défaut (sans paramètre)
- plusieurs constructeurs avec paramètres
- un constructeur par copie (clonage)

## Constructeur par défaut

Deux possibilités :

- soit sans paramètre
- soit tous ses paramètres sont initialisés par défaut

## Constructeur par copie

Ce constructeur permet de cloner un objet transmis en paramètre par une référence constante.

# Sur-définition des constructeurs

## Sur-définition

Il peut y avoir plusieurs constructeurs (sur-définition), ils doivent se distinguer par leur signature. Les plus courant :

- un constructeur par défaut (sans paramètre)
- plusieurs constructeurs avec paramètres
- un constructeur par copie (clonage)

## Constructeur par défaut

Deux possibilités :

- soit sans paramètre
- soit tous ses paramètres sont initialisés par défaut

## Constructeur par copie

Ce constructeur permet de cloner un objet transmis en paramètre par une référence constante.

# Synthèse des constructeurs

## Constructeurs synthétisés par le compilateur

- le constructeur par défaut est synthétisé par le compilateur s'il n'existe pas.
- le constructeur par copie est aussi synthétisé s'il n'est pas explicitement défini par le concepteur de la classe.

## Attention

les constructeurs synthétisés ne fonctionnent pas comme on le souhaiterait notamment dans le cas d'attribut nécessitant une allocation dynamique

# Synthèse des constructeurs

## Constructeurs synthétisés par le compilateur

- le constructeur par défaut est synthétisé par le compilateur s'il n'existe pas.
- le constructeur par copie est aussi synthétisé s'il n'est pas explicitement défini par le concepteur de la classe.

## Attention

les constructeurs synthétisés ne fonctionnent pas comme on le souhaiterait notamment dans le cas d'attribut nécessitant une allocation dynamique

# Constructeurs : exemple

## point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
// methodes
//constructeur par defaut
  Point();
//constructeur avec
    parametres
  Point(int,int);
//constructeur par copie
  Point(const Point &);
  ...
  };
```

## point.cc (C)

```
Point::Point()
{
x=0;
y=0;
}
Point::Point(int a, int b)
{
x=a;
y=b;
}
Point::Point(const Point &P)
{
x=P.x;
y=P.y;
}
```

## main.cc (U)

```
int main ()
{
Point P,Q;
Point R(2,3);
Point T(R);
Point U=T;
return 0;
}
```

# Constructeurs : exemple

## point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
// methodes
//constructeur par defaut
  Point();
//constructeur avec
    parametres
  Point(int,int);
//constructeur par copie
  Point(const Point &);
  ...
  };
```

## point.cc (C)

```
Point::Point()
{
x=0;
y=0;
}
Point::Point(int a, int b)
{
x=a;
y=b;
}
Point::Point(const Point &P)
{
x=P.x;
y=P.y;
}
```

## main.cc (U)

```
int main ()
{
Point P,Q;
Point R(2,3);
Point T(R);
Point U=T;
return 0;
}
```

# Constructeurs : exemple

## point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
// methodes
//constructeur par defaut
  Point();
//constructeur avec
    parametres
  Point(int,int);
//constructeur par copie
  Point(const Point &);
  ...
  };
```

## point.cc (C)

```
Point::Point()
{
x=0;
y=0;
}
Point::Point(int a, int b)
{
x=a;
y=b;
}
Point::Point(const Point &P)
{
x=P.x;
y=P.y;
}
```

## main.cc (U)

```
int main ()
{
Point P,Q;
Point R(2,3);
Point T(R);
Point U=T;
return 0;
}
```

# Constructeurs : exemple


class
Point/constructeur.cc

constructeur.cc (U)

```
int main ()
{
//appel du constructeur par defaut
Point P, Q;

//appel du constructeur avec parametres
Point R(2,3);

//appel du constructeur par copie
Point T(R), U=T;
return 0;
}
```

# Constructeurs : exemple


class
Point/constructeur.cc

## constructeur.cc (U)

```
int main ()
{
//appel du constructeur par defaut
Point P, Q;

//appel du constructeur avec parametres
Point R(2,3);

//appel du constructeur par copie
Point T(R), U=T;
return 0;
}
```

# Initialisation $\neq$ affectation

## Initialisation

C'est la création d'un objet avec appel d'un constructeur au moment de la définition/déclaration d'un objet

## Affectation

C'est la copie champs par champs des attributs dans le corps du programme, changement de la valeur des attributs d'un objet déjà créé


class
Point/affectation.cc

Constructeur par copie XOR affectation

```
int main ()
Point P(0,1), Q(2,3), R;
Point S=Q;// appel du constructeur par copie
R=P;//affectation
return 0;
}
```

# Initialisation $\neq$ affectation

## Initialisation

C'est la création d'un objet avec appel d'un constructeur au moment de la définition/déclaration d'un objet

## Affectation

C'est la copie champs par champs des attributs dans le corps du programme, changement de la valeur des attributs d'un objet déjà créé


class
Point/affectation.cc

# Initialisation $\neq$ affectation

## Initialisation

C'est la création d'un objet avec appel d'un constructeur au moment de la définition/déclaration d'un objet

## Affectation

C'est la copie champs par champs des attributs dans le corps du programme, changement de la valeur des attributs d'un objet déjà créé


`class Point/affectation.cc`

# Initialisation $\neq$ affectation

## Initialisation

C'est la création d'un objet avec appel d'un constructeur au moment de la définition/déclaration d'un objet

## Affectation

C'est la copie champs par champs des attributs dans le corps du programme, changement de la valeur des attributs d'un objet déjà créé


class
Point/affectation.cc

## constructeur par copie XOR affectation

```
int main (){
Point P(0,1), Q(2,3), R;
Point S=Q;// appel du constructeur par copie
S=P;//affectation
return 0;
}
```

Section 4

# Destructeur

# Destructeur

## Définition

Le destructeur est une méthode spéciale

- qui porte le même nom que la classe et précédée de `~`
- qui n'a pas de retour, même pas `void`
- qui est unique

## Appel du destructeur

Le destructeur est

- automatiquement appelé par le compilateur lorsqu'un objet est détruit, juste avant que la mémoire occupé par l'objet soit récupérée par le système.
- synthétisé par le compilateur s'il n'existe pas

# Destructeur

## Définition

Le destructeur est une méthode spéciale

- qui porte le même nom que la classe et précédée de `~`
- qui n'a pas de retour, même pas `void`
- qui est unique

## Appel du destructeur

Le destructeur est

- automatiquement appelé par le compilateur lorsqu'un objet est détruit, juste avant que la mémoire occupé par l'objet soit récupérée par le système.
- synthétisé par le compilateur s'il n'existe pas

# Destructeur

## Rôle du destructeur

Libération de l'espace mémoire qui a été alloué lors de la création de l'objet (appel du constructeur).

## Attention

Dans la cas d'attributs alloués dynamiquement, la restitution de la mémoire se fait par le destructeur qui doit être explicitement défini par le concepteur de la classe

# Destructeur

## Rôle du destructeur

Libération de l'espace mémoire qui a été alloué lors de la création de l'objet (appel du constructeur).

## Attention

Dans la cas d'attributs alloués dynamiquement, la restitution de la mémoire se fait par le destructeur qui doit être explicitement défini par le concepteur de la classe

# Destructeur : exemple avec allocation statique

## point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
...
//destructeur
 ~Point();
  };
```

## point.cc (C)

```
Point::~Point()
{
// rien a faire si pas d'allocation
    dynamique
}
```

# Destructeur : exemple avec allocation statique

## point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
...
//destructeur
 ~Point();
  };
```

## point.cc (C)

```
Point::~Point()
{
// rien a faire si pas d'allocation
    dynamique
}
```

Section 5

# Appel des constructeurs et destructeur

# Durée de vie et visibilité des objets

## Objet global

- Durée de vie : à partir de son allocation jusqu'à la fin du programme.
- Visibilité : à partir de sa déclaration jusqu'à la fin du programme ;

## Objet local

- Durée de vie :
  - Un objet local alloué statiquement reste alloué jusqu'à la fin du bloc où il est défini.
  - Un objet local alloué dynamiquement reste alloué jusqu'à l'appel de `delete`
- Visibilité : bloc où il est défini

## Objet `static`

- Un objet global déclaré `static` a une visibilité limitée au module
- Un objet local déclaré `static` a une visibilité limitée au bloc
- Dans les deux cas, la durée de vie commence à partir de sa déclaration jusqu'à la fin du programme.

# Durée de vie et visibilité des objets

## Objet global

- Durée de vie : à partir de son allocation jusqu'à la fin du programme.
- Visibilité : à partir de sa déclaration jusqu'à la fin du programme ;

## Objet local

- Durée de vie :
  - Un objet local alloué statiquement reste alloué jusqu'à la fin du bloc où il est défini.
  - Un objet local alloué dynamiquement reste alloué jusqu'à l'appel de `delete`
- Visibilité : bloc où il est défini

## Objet `static`

- Un objet global déclaré `static` a une visibilité limitée au module
- Un objet local déclaré `static` a une visibilité limitée au bloc
- Dans les deux cas, la durée de vie commence à partir de sa déclaration jusqu'à la fin du programme.

# Durée de vie et visibilité des objets

## Objet global

- Durée de vie : à partir de son allocation jusqu'à la fin du programme.
- Visibilité : à partir de sa déclaration jusqu'à la fin du programme ;

## Objet local

- Durée de vie :
  - Un objet local alloué statiquement reste alloué jusqu'à la fin du bloc où il est défini.
  - Un objet local alloué dynamiquement reste alloué jusqu'à l'appel de `delete`
- Visibilité : bloc où il est défini

## Objet `static`

- Un objet global déclaré `static` a une visibilité limitée au module
- Un objet local déclaré `static` a une visibilité limitée au bloc
- Dans les deux cas, la durée de vie commence à partir de sa déclaration jusqu'à la fin du programme.

# Constructeur/Destructeur : appel implicite et explicite

## Appels du constructeur

- explicite lors de la définition d'un objet (instance de la classe) et de son initialisation
- implicite pour les appels de fonctions/méthodes avec les arguments formels et avec le retour si ce sont des objets

## point.cc (C) : méthode Somme

```
Point Point::Somme(Point P){
//retourne le point somme de l'objet courant et du point P
Point S;
S.x = x + P.x;
S.y = y + P.y;
return S;
}
//{return Point (x+P.x,y+P.y);}
```

# Constructeur/Destructeur : appel implicite et explicite

## Appels du constructeur

- explicite lors de la définition d'un objet (instance de la classe) et de son initialisation
- implicite pour les appels de fonctions/méthodes avec les arguments formels et avec le retour si ce sont des objets

## point.cc (C) : méthode Somme

```
Point Point::Somme(Point P){
//retourne le point somme de l'objet courant et du point P
Point S;
S.x = x + P.x;
S.y = y + P.y;
return S;
}
//{return Point (x+P.x,y+P.y);}
```

# Constructeur/Destructeur : appel implicite et explicite

## main.cc (U)

```
int main()
{
Point A, B(2,2), C;
C = A.Somme(B);
C.Affiche();
(A.Somme(B)).Affiche
    ();
return 0;
}
```

## Appel de constructeur/destructeur

1. Appel explicite du const. par défaut pour A
2. Appel explicite du const. avec paramètres pour B
3. Appel explicite du const. par défaut pour C
4. Appel de A.Somme(B)
5. Affectation de la valeur de retour à C
6. ...

# Constructeur/Destructeur : appel implicite et explicite

## main.cc (U)

```
int main()
{
Point A, B(2,2), C;
C = A.Somme(B);
C.Affiche();
(A.Somme(B)).Affiche
    ();
return 0;
}
```

## Appel de constructeur/destructeur

1. Appel explicite du const. par défaut pour `A`
2. Appel explicite du const. avec paramètres pour `B`
3. Appel explicite du const. par défaut pour `C`
4. Appel de `A.Somme(B)`
   1. Appel (implicite) du const. par copie pour le paramètre `P` avec clonage de `B` (`Point P=B`)
   2. Appel (explicite) du const. par défaut pour `S`
   3. Appel (implicite) du const. par copie pour le retour de la fonction avec clonage de `S`
   4. Appel (implicite) du destructeur pour `S`
   5. Appel (implicite) du destructeur pour `P`
5. Affectation de la valeur de retour à `C`
6. ...

# Constructeur/Destructeur : appel implicite et explicite

## main.cc (U)

```
int main()
{
Point A, B(2,2), C;
C = A.Somme(B);
C.Affiche();
(A.Somme(B)).Affiche
    ();
return 0;
}
```

## Appel de constructeur/destructeur

1. Appel explicite du const. par défaut pour `A`
2. Appel explicite du const. avec paramètres pour `B`
3. Appel explicite du const. par défaut pour `C`
4. Appel de `A.Somme(B)`
   1. Appel (implicite) du const. par copie pour le paramètre `P` avec clonage de `B` (`Point P=B`)
   2. Appel (explicite) du const. par défaut pour `S`
   3. Appel (implicite) du const. par copie pour le retour de la fonction avec clonage de `S`
   4. Appel (implicite) du destructeur pour `S`
   5. Appel (implicite) du destructeur pour `P`
5. Affectation de la valeur de retour à `C`
6. ...

# Constructeur/Destructeur : appel implicite et explicite

## main.cc (U)

```
int main()
{
Point A, B(2,2), C;
C = A.Somme(B);
C.Affiche();
(A.Somme(B)).Affiche
    ();
return 0;
}
```

## Appel de constructeur/destructeur

1. Appel explicite du const. par défaut pour `A`
2. Appel explicite du const. avec paramètres pour `B`
3. Appel explicite du const. par défaut pour `C`
4. Appel de `A.Somme(B)`
   1. Appel (implicite) du const. par copie pour le paramètre `P` avec clonage de `B` (`Point P=B`)
   2. Appel (explicite) du const. par défaut pour `S`
   3. Appel (implicite) du const. par copie pour le retour de la fonction avec clonage de `S`
   4. Appel (implicite) du destructeur pour `S`
   5. Appel (implicite) du destructeur pour `P`
5. Affectation de la valeur de retour à `C`
6. ...

# Constructeur/Destructeur : appel implicite et explicite

## main.cc (U)

```
int main()
{
Point A, B(2,2), C;
C = A.Somme(B);
C.Affiche();
(A.Somme(B)).Affiche
    ();
return 0;
}
```

## Appel de constructeur/destructeur

1. Appel explicite du const. par défaut pour `A`
2. Appel explicite du const. avec paramètres pour `B`
3. Appel explicite du const. par défaut pour `C`
4. Appel de `A.Somme(B)`
   1. Appel (implicite) du const. par copie pour le paramètre `P` avec clonage de `B` (`Point P=B`)
   2. Appel (explicite) du const. par défaut pour `S`
   3. Appel (implicite) du const. par copie pour le retour de la fonction avec clonage de `S`
   4. Appel (implicite) du destructeur pour `S`
   5. Appel (implicite) du destructeur pour `P`
5. Affectation de la valeur de retour à `C`
6. ...

# Constructeur/Destructeur : appel implicite et explicite

## main.cc (U)

```
int main()
{
Point A, B(2,2), C;
C = A.Somme(B);
C.Affiche();
(A.Somme(B)).Affiche
    ();
return 0;
}
```

## Appel de constructeur/destructeur

1. Appel explicite du const. par défaut pour `A`
2. Appel explicite du const. avec paramètres pour `B`
3. Appel explicite du const. par défaut pour `C`
4. Appel de `A.Somme(B)`
   1. Appel (implicite) du const. par copie pour le paramètre `P` avec clonage de `B` (`Point P=B`)
   2. Appel (explicite) du const. par défaut pour `S`
   3. Appel (implicite) du const. par copie pour le retour de la fonction avec clonage de `S`
   4. Appel (implicite) du destructeur pour `S`
   5. Appel (implicite) du destructeur pour `P`
5. Affectation de la valeur de retour à `C`
6. ...

# Constructeur/Destructeur : appel implicite et explicite

## main.cc (U)

```
int main()
{
Point A, B(2,2), C;
C = A.Somme(B);
C.Affiche();
(A.Somme(B)).Affiche
    ();
return 0;
}
```

## Appel de constructeur/destructeur

1. Appel explicite du const. par défaut pour `A`
2. Appel explicite du const. avec paramètres pour `B`
3. Appel explicite du const. par défaut pour `C`
4. Appel de `A.Somme(B)`
   1. Appel (implicite) du const. par copie pour le paramètre `P` avec clonage de `B` (`Point P=B`)
   2. Appel (explicite) du const. par défaut pour `S`
   3. Appel (implicite) du const. par copie pour le retour de la fonction avec clonage de `S`
   4. Appel (implicite) du destructeur pour S
   5. Appel (implicite) du destructeur pour P
5. Affectation de la valeur de retour à `C`
6. ...

# Constructeur/Destructeur : appel implicite et explicite

## main.cc (U)

```
int main()
{
Point A, B(2,2), C;
C = A.Somme(B);
C.Affiche();
(A.Somme(B)).Affiche
    ();
return 0;
}
```

## Appel de constructeur/destructeur

1. Appel explicite du const. par défaut pour `A`
2. Appel explicite du const. avec paramètres pour `B`
3. Appel explicite du const. par défaut pour `C`
4. Appel de `A.Somme(B)`
   1. Appel (implicite) du const. par copie pour le paramètre `P` avec clonage de `B` (`Point P=B`)
   2. Appel (explicite) du const. par défaut pour `S`
   3. Appel (implicite) du const. par copie pour le retour de la fonction avec clonage de `S`
   4. Appel (implicite) du destructeur pour S
   5. Appel (implicite) du destructeur pour P
5. Affectation de la valeur de retour à `C`
6. ...

# Constructeur/Destructeur : appel implicite et explicite

## main.cc (U)

```
int main()
{
Point A, B(2,2), C;
C = A.Somme(B);
C.Affiche();
(A.Somme(B)).Affiche
    ();
return 0;
}
```

## Appel de constructeur/destructeur

1. Appel explicite du const. par défaut pour `A`
2. Appel explicite du const. avec paramètres pour `B`
3. Appel explicite du const. par défaut pour `C`
4. Appel de `A.Somme(B)`
   1. Appel (implicite) du const. par copie pour le paramètre `P` avec clonage de `B` (`Point P=B`)
   2. Appel (explicite) du const. par défaut pour `S`
   3. Appel (implicite) du const. par copie pour le retour de la fonction avec clonage de `S`
   4. Appel (implicite) du destructeur pour S
   5. Appel (implicite) du destructeur pour P
5. Affectation de la valeur de retour à `C`
6. ...

# Constructeur/Destructeur : appel implicite et explicite

## main.cc (U)

```
int main()
{
Point A, B(2,2), C;
C = A.Somme(B);
C.Affiche();
(A.Somme(B)).Affiche
    ();
return 0;
}
```

## Appel de constructeur/destructeur

1. Appel explicite du const. par défaut pour `A`
2. Appel explicite du const. avec paramètres pour `B`
3. Appel explicite du const. par défaut pour `C`
4. Appel de `A.Somme(B)`
   1. Appel (implicite) du const. par copie pour le paramètre `P` avec clonage de `B` (`Point P=B`)
   2. Appel (explicite) du const. par défaut pour `S`
   3. Appel (implicite) du const. par copie pour le retour de la fonction avec clonage de `S`
   4. Appel (implicite) du destructeur pour S
   5. Appel (implicite) du destructeur pour P
5. Affectation de la valeur de retour à `C`
6. ...

# Constructeur/Destructeur : appel implicite et explicite

## main.cc (U)

```
int main()
{
Point A, B(2,2), C;
C = A.Somme(B);
C.Affiche();
(A.Somme(B)).Affiche
    ();
return 0;
}
```

## Appel de constructeur/destructeur

1. Appel explicite du const. par défaut pour `A`
2. Appel explicite du const. avec paramètres pour `B`
3. Appel explicite du const. par défaut pour `C`
4. Appel de `A.Somme(B)`
   1. Appel (implicite) du const. par copie pour le paramètre `P` avec clonage de `B` (`Point P=B`)
   2. Appel (explicite) du const. par défaut pour `S`
   3. Appel (implicite) du const. par copie pour le retour de la fonction avec clonage de `S`
   4. Appel (implicite) du destructeur pour S
   5. Appel (implicite) du destructeur pour P
5. Affectation de la valeur de retour à `C`
6. ...

# Constructeur/Destructeur : appel implicite et explicite

## Exercice

Lister les appels aux constructeurs et destructeur dans l'exemple suivant :

### point.cc (C)

```
Point Point::Somme(Point P)
{
return Point (x+P.x,y+P.y);
}
```

### main.cc (U)

```
int main()
{
Point A, B(2,2), C;
C = A.Somme(B);
C.Affiche();
(A.Somme(B)).Affiche();
return 0;
}
```

Section 6

# Allocation dynamique

- Constructeur par copie

# Attribut, constructeur et allocation dynamique

## Allocation dynamique

Si un attribut est de type pointeur alors le concepteur de la classe doit faire

1. l'allocation mémoire dans les constructeurs
2. la libération de l'espace mémoire dans le destructeur

## class TabEntier

```
class TabEntier{
private :
   int taille;
   int* tab;
public:
   TabEntier();
   TabEntier(int,int*);
   TabEntier(const TabEntier &);
   ~TabEntier();
};
```

# Attribut, constructeur et allocation dynamique

## Constructeur par défaut

```
TabEntier::TabEntier ()
{
taille = 0;
tab = 0;// pointeur NULL
}
```

## Constructeur avec param.

```
TabEntier::TabEntier (int n, int * t)
{
taille = n;

//allocation
tab = new int[taille];

// initialisation
for (int i=0;i<taille;i++)
   tab[i] = t[i];
}
```

# Attribut, constructeur et allocation dynamique

## Constructeur par défaut

```
TabEntier::TabEntier ()
{
taille = 0;
tab = 0;// pointeur NULL
}
```

## Constructeur avec param.

```
TabEntier::TabEntier (int n, int * t)
{
taille = n;

//allocation
tab = new int[taille];

// initialisation
for (int i=0;i<taille;i++)
   tab[i] = t[i];
}
```

# Attribut, constructeur et allocation dynamique

## Constructeur par copie

```
TabEntier::TabEntier (const TabEntier & T)
{
taille = T.taille;

//allocation
tab = new int[taille];

// initialisation/recopie
for (int i=0;i<taille;i++)
   tab[i] = T.tab[i];
}
```

## Destructeur

```
TabEntier::~TabEntier ()
{
// desallocation
    uniquement de l'
    espace alloue
    dynamiquement
delete [] tab;
}
```

# Attribut, constructeur et allocation dynamique

## Constructeur par copie

```
TabEntier::TabEntier (const TabEntier & T)
{
taille = T.taille;

//allocation
tab = new int[taille];

// initialisation/recopie
for (int i=0;i<taille;i++)
   tab[i] = T.tab[i];
}
```

## Destructeur

```
TabEntier::~TabEntier ()
{
// desallocation
    uniquement de l'
    espace alloue
    dynamiquement
delete [] tab;
}
```

# Synthèse du constructeur par copie

## Constructeur par copie absent

Le compilateur synthétise le constructeur par copie en faisant une copie champs à champs des attributs

## Exemple de classe

```
class Pint{
private :
  int* i;
public :
  Pint();
  Pint(int);
  ~Pint();
};
```

## Définition des constructeurs/destructeur

```
Pint::Pint(){//constructeur par defaut
i=0;//allocation pointeur nul
}
Pint::Pint(int a){//constructeur avec param
i= new int;// allocation
*i=a;//initialisation
}
Pint::~Pint(){//destructeur
delete i;//desallocation
}
```

# Synthèse du constructeur par copie

## Constructeur par copie absent

Le compilateur synthétise le constructeur par copie en faisant une copie champs à champs des attributs

## Exemple de classe

```
class Pint{
private :
  int* i;
public :
  Pint();
  Pint(int);
  ~Pint();
};
```

## Définition des constructeurs/destructeur

```
Pint::Pint(){//constructeur par defaut
i=0;//allocation pointeur nul
}
Pint::Pint(int a){//constructeur avec param
i= new int;// allocation
*i=a;//initialisation
}
Pint::~Pint(){//destructeur
delete i;//desallocation
}
```

# Synthèse du constructeur par copie

## Programme

```
int main()
{
Pint x,y(2),z(y);

}
```

## Définition des constructeurs

```
Pint::Pint()//constructeur par defaut
{
i=0;//pointeur nul
}
Pint::Pint(int a)//constructeur avec param
{
i= new int;// allocation
*i=a;//initialisation
}
Pint::~Pint()
{delete i;}
```

# Synthèse du constructeur par copie

## Programme

```
int main()
{
Pint x,y(2),z(y);

}
```

## Définition des constructeurs

```
Pint::Pint()//constructeur par defaut
{
i=0;//pointeur nul
}
Pint::Pint(int a)//constructeur avec param
{
i= new int;// allocation
*i=a;//initialisation
}
Pint::~Pint()
{delete i;}
```

# Synthèse du constructeur par copie

## Constructeur par copie synthétisé par le compilateur

```
Pint::Pint(const Pint &p)

{
i=p.i;
}
```

## Programme

```
int main()
{
Pint x,y(2),z(y);

}
```

## Exercice

Simuler l'état de la mémoire pour l'exécution du programme. Quel problème se pose ?

# Synthèse du constructeur par copie

## Constructeur par copie synthétisé par le compilateur

```
Pint::Pint(const Pint &p)

{
i=p.i;
}
```

## Programme

```
int main()
{
Pint x,y(2),z(y);

}
```

## Exercice

Simuler l'état de la mémoire pour l'exécution du programme. Quel problème se pose ?

# Synthèse du constructeur par copie

## Constructeur par copie synthétisé par le compilateur

```
Pint::Pint(const Pint &p)

{
i=p.i;
}
```

## Programme

```
int main()
{
Pint x,y(2),z(y);

}
```

## Exercice

Simuler l'état de la mémoire pour l'exécution du programme. Quel problème se pose ?

# Le bon constructeur par copie

## Classe

```
class Pint{
private :
  int*i;
public :
  Pint();
  Pint(int);
  Pint(const Pint&);
  ~Pint();
};
```

## Le constructeur par copie

```
Pint::Pint(const Pint &p)
{//constructeur par copie
i= new int;//allocation
*i=*(p.i);//initialisation
}
```

## Exécution

```
int main()
{
Pint x,y(2),z(y);
return 0;
}
```

## Exercice

Simuler l'état de la mémoire pour l'exécution du programme. Faire le point.

# Le bon constructeur par copie

## Classe

```
class Pint{
private :
  int*i;
public :
  Pint();
  Pint(int);
  Pint(const Pint&);
  ~Pint();
};
```

## Le constructeur par copie

```
Pint::Pint(const Pint &p)
{//constructeur par copie
i= new int;//allocation
*i=*(p.i);//initialisation
}
```

## Exécution

```
int main()
{
Pint x,y(2),z(y);
return 0;
}
```

## Exercice

Simuler l'état de la mémoire pour l'exécution du programme. Faire le point.

# Le bon constructeur par copie

## Classe

```
class Pint{
private :
  int*i;
public :
  Pint();
  Pint(int);
  Pint(const Pint&);
  ~Pint();
};
```

## Le constructeur par copie

```
Pint::Pint(const Pint &p)
{//constructeur par copie
i= new int;//allocation
*i=*(p.i);//initialisation
}
```

## Exécution

```
int main()
{
Pint x,y(2),z(y);
return 0;
}
```

## Exercice

Simuler l'état de la mémoire pour l'exécution du programme. Faire le point.

# Le bon constructeur par copie

## Classe

```
class Pint{
private :
  int*i;
public :
  Pint();
  Pint(int);
  Pint(const Pint&);
  ~Pint();
};
```

## Le constructeur par copie

```
Pint::Pint(const Pint &p)
{//constructeur par copie
i= new int;//allocation
*i=*(p.i);//initialisation
}
```

## Exécution

```
int main()
{
Pint x,y(2),z(y);
return 0;
}
```

## Exercice

Simuler l'état de la mémoire pour l'exécution du programme. Faire le point.

Section 7

# Forme canonique de Coplien

# Forme canonique de Coplien

## La forme canonique de Coplien

La forme canonique de Coplien est définie par

- le constructeur par défaut
- le constructeur par copie
- le destructeur
- la surcharge de l'opérateur d'affectation

# Forme canonique de Coplien

## La forme canonique de Coplien

La forme canonique de Coplien est définie par

- le constructeur par défaut
- le constructeur par copie
- le destructeur
- la surcharge de l'opérateur d'affectation

# Forme canonique de Coplien

## La forme canonique de Coplien

La forme canonique de Coplien est définie par

- le constructeur par défaut
- le constructeur par copie
- le destructeur
- la surcharge de l'opérateur d'affectation

# Forme canonique de Coplien

## La forme canonique de Coplien

La forme canonique de Coplien est définie par

- le constructeur par défaut
- le constructeur par copie
- le destructeur
- la surcharge de l'opérateur d'affectation

# Forme canonique de Coplien

## Pour une classe A

```
class A{
private :
 //attributs
 ...
public:
 A(); // constructeur par defaut
 A(const A&); // constructeur par copie
 ~A(); // destructeur
 A& operator = (const A&);// surcharge de l'oprateur d'affectation
 ...
 };
```

# Forme canonique de Coplien

## Attribut dynamique

- les constructeurs allouent l'espace mémoire et le destructeur désalloue
- les éléments suivants sont indispensables :
  - les constructeurs par copie et par défaut
  - le destructeur
  - la surcharge de l'opérateur d'affectation

## Allocation dynamique

L'affectation de deux objets posent des problèmes similaires au constructeur par copie si elle est synthétisée par le compilateur : partage de la même zone mémoire.

# Forme canonique de Coplien

## Attribut dynamique

- les constructeurs allouent l'espace mémoire et le destructeur désalloue
- les éléments suivants sont indispensables :
  - les constructeurs par copie et par défaut
  - le destructeur
  - la surcharge de l'opérateur d'affectation

## Allocation dynamique

L'affectation de deux objets posent des problèmes similaires au constructeur par copie si elle est synthétisée par le compilateur : partage de la même zone mémoire.

# Forme canonique de Coplien

## Attribut dynamique

- les constructeurs allouent l'espace mémoire et le destructeur désalloue
- les éléments suivants sont indispensables :
  - les constructeurs par copie et par défaut
  - le destructeur
  - la surcharge de l'opérateur d'affectation

## Allocation dynamique

L'affectation de deux objets posent des problèmes similaires au constructeur par copie si elle est synthétisée par le compilateur : partage de la même zone mémoire.

# Forme canonique de Coplien

## Attribut dynamique

- les constructeurs allouent l'espace mémoire et le destructeur désalloue
- les éléments suivants sont indispensables :
  - les constructeurs par copie et par défaut
  - le destructeur
  - la surcharge de l'opérateur d'affectation

## Allocation dynamique

L'affectation de deux objets posent des problèmes similaires au constructeur par copie si elle est synthétisée par le compilateur : partage de la même zone mémoire.

# Forme canonique de Coplien

## Attribut dynamique

- les constructeurs allouent l'espace mémoire et le destructeur désalloue
- les éléments suivants sont indispensables :
  - les constructeurs par copie et par défaut
  - le destructeur
  - la surcharge de l'opérateur d'affectation

## Allocation dynamique

L'affectation de deux objets posent des problèmes similaires au constructeur par copie si elle est synthétisée par le compilateur : partage de la même zone mémoire.

# Forme canonique de Coplien

## Attribut dynamique

- les constructeurs allouent l'espace mémoire et le destructeur désalloue
- les éléments suivants sont indispensables :
  - les constructeurs par copie et par défaut
  - le destructeur
  - la surcharge de l'opérateur d'affectation

## Allocation dynamique

L'affectation de deux objets posent des problèmes similaires au constructeur par copie si elle est synthétisée par le compilateur : partage de la même zone mémoire.

Section 8

# Attributs statiques

# Attribut

## Allocation mémoire

Chaque objet (instance de classe) possède une zone mémoire correspondant à ses attributs allouée lors de l'appel du constructeur

### Classe

```
class Point{
private :
  int x,y;
...
};
```

### Programme

```
int main()
{
Point P,Q(2,3),R(Q);
return 0;
}
```

### Zone mémoire

Les zones mémoires sont toutes distinctes :

- l'objet `P` est composé de deux `int` désignés par `P.x` et `P.y`
- Idem pour `Q` et `R`

# Attribut

## Allocation mémoire

Chaque objet (instance de classe) possède une zone mémoire correspondant à ses attributs allouée lors de l'appel du constructeur

### Classe

```
class Point{
private :
  int x,y;
...
};
```

### Programme

```
int main()
{
Point P,Q(2,3),R(Q);
return 0;
}
```

### Zone mémoire

Les zones mémoires sont toutes distinctes :

- l'objet `P` est composé de deux `int` désignés par `P.x` et `P.y`
- Idem pour `Q` et `R`

# Attribut

## Allocation mémoire

Chaque objet (instance de classe) possède une zone mémoire correspondant à ses attributs allouée lors de l'appel du constructeur

### Classe

```
class Point{
private :
  int x,y;
...
};
```

### Programme

```
int main()
{
Point P,Q(2,3),R(Q);
return 0;
}
```

## Zone mémoire

Les zones mémoires sont toutes distinctes :

- l'objet `P` est composé de deux `int` désignés par `P.x` et `P.y`
- Idem pour `Q` et `R`

# Attribut `static`

## Syntaxe

```
class A{
private:
  static type ident;
};
```

## Sémantique

L'attribut qualifié de `static` est commun à tous les objets de la classe

- cet attribut ne peut pas être initialisé par un constructeur ou dans sa déclaration
- cet attribut doit être initialisé en dehors de la classe

## Remarque

Sorte d'attribut global à la classe dans l'espace de visibilité de la classe.

# Attribut `static`

## Syntaxe

```
class A{
private:
  static type ident;
};
```

## Sémantique

L'attribut qualifié de `static` est commun à tous les objets de la classe

- cet attribut ne peut pas être initialisé par un constructeur ou dans sa déclaration
- cet attribut doit être initialisé en dehors de la classe

## Remarque

Sorte d'attribut global à la classe dans l'espace de visibilité de la classe.

# Attribut `static`

## Syntaxe

```
class A{
private:
  static type ident;
};
```

## Sémantique

L'attribut qualifié de `static` est commun à tous les objets de la classe

- cet attribut ne peut pas être initialisé par un constructeur ou dans sa déclaration
- cet attribut doit être initialisé en dehors de la classe

## Remarque

Sorte d'attribut global à la classe dans l'espace de visibilité de la classe.

# Attribut `static` : exemple

## Classe

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

## Programme

```
int main()
{
Exemple A,B;
return 0;
}
```

## partage d'attribut

- `A.n` et `B.n` désignent le même espace mémoire de taille `int`
- `A.x` et `B.x` désignent deux zones mémoires distinctes

# Attribut `static` : exemple

## Classe

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

## Programme

```
int main()
{
Exemple A,B;
return 0;
}
```

## partage d'attribut

- `A.n` et `B.n` désignent le même espace mémoire de taille `int`
- `A.x` et `B.x` désignent deux zones mémoires distinctes

# Attribut `static` : exemple

## Classe

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

## Programme

```
int main()
{
Exemple A,B;
return 0;
}
```

## partage d'attribut

- `A.n` et `B.n` désignent le même espace mémoire de taille `int`
- `A.x` et `B.x` désignent deux zones mémoires distinctes

# Attribut `static` : à quoi ça sert ?

A compter le nombre d'instances de la classe...

CpteObj.h

```
class CpteObj{
private :
  static int cpt;
public :
  CpteObj();
  ~CpteObj();
};
```

CpteObj.cc

```
int CpteObj::cpt=0;//initialisation
    cpt
CpteObj::CpteObj()
{
cout<<"const."<<++cpt<<"objets"<<endl;
}
CpteObj::~CpteObj()
{
cout<<"dest."<<--cpt<<"objets"<<endl;
}
```

main.cc

```
void f()
{CpteObj U,V;
}
int main()
{
void f();
CpteObj A;
f();
CpteObj B;
return 0;
}
```


`make CpteObj`

# Attribut `static` : à quoi ça sert ?

A compter le nombre d'instances de la classe...

**CpteObj.h**

```
class CpteObj{
private :
  static int cpt;
public :
  CpteObj();
  ~CpteObj();
};
```

**CpteObj.cc**

```
int CpteObj::cpt=0;//initialisation
    cpt
CpteObj::CpteObj()
{
cout<<"const."<<++cpt<<"objets"<<endl;
}
CpteObj::~CpteObj()
{
cout<<"dest."<<--cpt<<"objets"<<endl;
}
```

**main.cc**

```
void f()
{CpteObj U,V;
}
int main()
{
void f();
CpteObj A;
f();
CpteObj B;
return 0;
}
```


`make CpteObj`

# Attribut `static` : à quoi ça sert ?

A compter le nombre d'instances de la classe...

## CpteObj.h

```
class CpteObj{
private :
  static int cpt;
public :
  CpteObj();
  ~CpteObj();
};
```

## CpteObj.cc

```
int CpteObj::cpt=0;//initialisation
    cpt
CpteObj::CpteObj()
{
cout<<"const."<<++cpt<<"objets"<<endl;
}
CpteObj::~CpteObj()
{
cout<<"dest."<<--cpt<<"objets"<<endl;
}
```

## main.cc

```
void f()
{CpteObj U,V;
}
int main()
{
void f();
CpteObj A;
f();
CpteObj B;
return 0;
}
```


`make CpteObj`

# Attribut `static` : à quoi ça sert ?

A compter le nombre d'instances de la classe...

## CpteObj.h

```
class CpteObj{
private :
  static int cpt;
public :
  CpteObj();
  ~CpteObj();
};
```

## CpteObj.cc

```
int CpteObj::cpt=0;//initialisation
    cpt
CpteObj::CpteObj()
{
cout<<"const."<<++cpt<<"objets"<<endl;
}
CpteObj::~CpteObj()
{
cout<<"dest."<<--cpt<<"objets"<<endl;
}
```

## main.cc

```
void f()
{CpteObj U,V;
}
int main()
{
void f();
CpteObj A;
f();
CpteObj B;
return 0;
}
```


`make CpteObj`

Section 9

# Espace de visibilité et d'accessibilité

- Visibilité limitée à la classe
- Visibilité locale
- Local
- Module

# Espace de visibilité : limité à la classe

## Attribut privé

- Un attribut privé d'un objet n'est accessible/visible que par les méthodes de la classe.
- Des méthodes appelées accesseurs fournissent à l'utilisateur une interface permettant d'accéder aux attributs privés

point.h

```
class Point{
private :
  int x,y;
public:
...
int Get_Abscisse(){return x;}
int Get_Ordonnee(){return y;}
...
};
```

main.cc

```
int main()
{
Point P(2,3);
cout<<P.x; // KO l'attribut est
    private
cout<<P.Get_Abscisse(); // OK
cout<<P.Get_Ordonnee(); // OK
return 0;
}
```

# Espace de visibilité : limité à la classe

## Attribut privé

- Un attribut privé d'un objet n'est accessible/visible que par les méthodes de la classe.
- Des méthodes appelées accesseurs fournissent à l'utilisateur une interface permettant d'accéder aux attributs privés

point.h

```
class Point{
private :
  int x,y;
public:
...
int Get_Abscisse(){return x;}
int Get_Ordonnee(){return y;}
...
};
```

main.cc

```
int main()
{
Point P(2,3);
cout<<P.x; // KO l'attribut est
    private
cout<<P.Get_Abscisse(); // OK
cout<<P.Get_Ordonnee(); // OK
return 0;
}
```

# Espace de visibilité : limité à la classe

## Attribut privé

- Un attribut privé d'un objet n'est accessible/visible que par les méthodes de la classe.
- Des méthodes appelées accesseurs fournissent à l'utilisateur une interface permettant d'accéder aux attributs privés

### point.h

```
class Point{
private :
  int x,y;
public:
...
int Get_Abscisse(){return x;}
int Get_Ordonnee(){return y;}
...
};
```

### main.cc

```
int main()
{
Point P(2,3);
cout<<P.x; // KO l'attribut est
    private
cout<<P.Get_Abscisse(); // OK
cout<<P.Get_Ordonnee(); // OK
return 0;
}
```

# Accessibilité des attributs

## Attribut privé

- Un attribut privé d'un objet n'est modifiable que par les méthodes de la classe.
- Des méthodes appelées modificateurs permettent à l'utilisateur une interface permettant de modifier les attributs privés de l'objet.

point.h

```
class Point{
private :
  int x,y;
public:
...
void Put_Abscisse(int a){x=a;}
void Put_Ordonnee(int b){y=b;}
...
};
```

main.cc

```
int main()
{
Point P;
P.x=2;// KO P.x est private
P.y=3; // KO idem pour P.y
P.Put_Abscisse(2); // modificateur
P.Put_Ordonnee(3); // modificateur
return 0;
}
```

# Accessibilité des attributs

## Attribut privé

- Un attribut privé d'un objet n'est modifiable que par les méthodes de la classe.
- Des méthodes appelées modificateurs permettent à l'utilisateur une interface permettant de modifier les attributs privés de l'objet.

point.h

```
class Point{
private :
  int x,y;
public:
...
void Put_Abscisse(int a){x=a;}
void Put_Ordonnee(int b){y=b;}
...
};
```

main.cc

```
int main()
{
Point P;
P.x=2;// KO P.x est private
P.y=3; // KO idem pour P.y
P.Put_Abscisse(2); // modificateur
P.Put_Ordonnee(3); // modificateur
return 0;
}
```

# Accessibilité des attributs

## Attribut privé

- Un attribut privé d'un objet n'est modifiable que par les méthodes de la classe.
- Des méthodes appelées modificateurs permettent à l'utilisateur une interface permettant de modifier les attributs privés de l'objet.

### point.h

```
class Point{
private :
  int x,y;
public:
...
void Put_Abscisse(int a){x=a;}
void Put_Ordonnee(int b){y=b;}
...
};
```

### main.cc

```
int main()
{
Point P;
P.x=2;// KO P.x est private
P.y=3; // KO idem pour P.y
P.Put_Abscisse(2); // modificateur
P.Put_Ordonnee(3); // modificateur
return 0;
}
```

# Espace de visibilité : local à un bloc ou une fonction

## Objet statique

- Masque les identificateurs globaux
- L'opérateur de portée `::` permet
  - de désigner un objet global masqué par un local
  - de distinguer les attributs
  - d'utiliser des fonctions systèmes

### global ou local

```
int x = 10;// x global
void f()
{
int x = 1; // local qui masque le global
x++; // incrementation du local
::x++; // incrementation de x global
}
```

### Possible mais maladroit

```
class A{
  int m;
public :
  void Init(int m){A::m = m;}
};
```

# Espace de visibilité : local à un bloc ou une fonction

## Objet statique

- Masque les identificateurs globaux
- L'opérateur de portée `::` permet
  - de désigner un objet global masqué par un local
  - de distinguer les attributs
  - d'utiliser des fonctions systèmes

### global ou local

```
int x = 10;// x global
void f()
{
int x = 1; // local qui masque le global
x++; // incrementation du local
::x++; // incrementation de x global
}
```

### Possible mais maladroit

```
class A{
  int m;
public :
  void Init(int m){A::m = m;}
};
```

# Espace de visibilité : local à un bloc ou une fonction

## Objet statique

- Masque les identificateurs globaux
- L'opérateur de portée `::` permet
  - de désigner un objet global masqué par un local
  - de distinguer les attributs
  - d'utiliser des fonctions systèmes

### global ou local

```
int x = 10;// x global
void f()
{
int x = 1; // local qui masque le global
x++; // incrementation du local
::x++; // incrementation de x global
}
```

### Possible mais maladroit

```
class A{
  int m;
public :
  void Init(int m){A::m = m;}
};
```

# Espace de visibilité : local à un bloc ou une fonction

## Objet statique

- Masque les identificateurs globaux
- L'opérateur de portée `::` permet
  - de désigner un objet global masqué par un local
  - de distinguer les attributs
  - d'utiliser des fonctions systèmes

## global ou local

```
int x = 10;// x global
void f()
{
int x = 1; // local qui masque le global
x++; // incrementation du local
::x++; // incrementation de x global
}
```

## Possible mais maladroit

```
class A{
  int m;
public :
  void Init(int m){A::m = m;}
};
```

# Espace de visibilité : local à un bloc ou une fonction

## Objet statique

- Masque les identificateurs globaux
- L'opérateur de portée `::` permet
  - de désigner un objet global masqué par un local
  - de distinguer les attributs
  - d'utiliser des fonctions systèmes

### global ou local

```
int x = 10;// x global
void f()
{
int x = 1; // local qui masque le global
x++; // incrementation du local
::x++; // incrementation de x global
}
```

### Possible mais maladroit

```
class A{
  int m;
public :
  void Init(int m){A::m = m;}
};
```

# Espace de visibilité : local à un bloc ou une fonction

## Objet statique

- Masque les identificateurs globaux
- L'opérateur de portée `::` permet
  - de désigner un objet global masqué par un local
  - de distinguer les attributs
  - d'utiliser des fonctions systèmes

## global ou local

```
int x = 10;// x global
void f()
{
int x = 1; // local qui masque le global
x++; // incrementation du local
::x++; // incrementation de x global
}
```

## Possible mais maladroit

```
class A{
  int m;
public :
  void Init(int m){A::m = m;}
};
```

# Espace de visibilité : local à un bloc ou une fonction

## Objet statique

- Masque les identificateurs globaux
- L'opérateur de portée `::` permet
  - de désigner un objet global masqué par un local
  - de distinguer les attributs
  - d'utiliser des fonctions systèmes

## global ou local

```
int x = 10;// x global
void f()
{
int x = 1; // local qui masque le global
x++; // incrementation du local
::x++; // incrementation de x global
}
```

## Possible mais maladroit

```
class A{
  int m;
public :
  void Init(int m){A::m = m;}
};
```

# Espace de visibilité : local à un bloc ou une fonction

## Objet dynamique

Un objet alloué dynamiquement est visible jusqu'à sa désallocation

# Espace de visibilité : module

## Visibilité dans un module

- si l'objet est déclaré à l'extérieur des fonctions ou classe,
  - sa visibilité est globale
  - visibilité possible dans les autres modules où il doit y être déclaré `extern`
- visibilité limitée au module s'il est déclaré `static`

Section 10

# Méthodes

# Les méthodes

## Les méthodes sont des fonctions (membres)

Les méthodes ont toutes les propriétés des fonctions en C++

- Sur-définition (différenciation avec la signature)
- Arguments par défaut
- Définies `inline`
  - avec `inline` en dehors de la classe
  - ou sans le qualificatif `inline` dans la classe
- un objet peut être en argument d'une méthode
  - transmis par valeur
  - transmis par adresse
  - transmis par référence
  - transmis par référence constante
- le retour d'une méthode peut être
  - un objet
  - un pointeur sur un objet
  - une référence sur un objet

# Les méthodes

## Les méthodes sont des fonctions (membres)

Les méthodes ont toutes les propriétés des fonctions en C++

- Sur-définition (différenciation avec la signature)
- Arguments par défaut
- Définies `inline`
  - avec `inline` en dehors de la classe
  - ou sans le qualificatif `inline` dans la classe
- un objet peut être en argument d'une méthode
  - transmis par valeur
  - transmis par adresse
  - transmis par référence
  - transmis par référence constante
- le retour d'une méthode peut être
  - un objet
  - un pointeur sur un objet
  - une référence sur un objet

# Les méthodes

## Les méthodes sont des fonctions (membres)

Les méthodes ont toutes les propriétés des fonctions en C++

- Sur-définition (différenciation avec la signature)
- Arguments par défaut
- Définies `inline`
  - avec `inline` en dehors de la classe
  - ou sans le qualificatif `inline` dans la classe
- un objet peut être en argument d'une méthode
  - transmis par valeur
  - transmis par adresse
  - transmis par référence
  - transmis par référence constante
- le retour d'une méthode peut être
  - un objet
  - un pointeur sur un objet
  - une référence sur un objet

# Les méthodes

## Les méthodes sont des fonctions (membres)

Les méthodes ont toutes les propriétés des fonctions en C++

- Sur-définition (différenciation avec la signature)
- Arguments par défaut
- Définies `inline`
  - avec `inline` en dehors de la classe
  - ou sans le qualificatif `inline` dans la classe
- un objet peut être en argument d'une méthode
  - transmis par valeur
  - transmis par adresse
  - transmis par référence
  - transmis par référence constante
- le retour d'une méthode peut être
  - un objet
  - un pointeur sur un objet
  - une référence sur un objet

# Les méthodes

## Les méthodes sont des fonctions (membres)

Les méthodes ont toutes les propriétés des fonctions en C++

- Sur-définition (différenciation avec la signature)
- Arguments par défaut
- Définies `inline`
  - avec `inline` en dehors de la classe
  - ou sans le qualificatif `inline` dans la classe
- un objet peut être en argument d'une méthode
  - transmis par valeur
  - transmis par adresse
  - transmis par référence
  - transmis par référence constante
- le retour d'une méthode peut être
  - un objet
  - un pointeur sur un objet
  - une référence sur un objet

# Les méthodes

## Les méthodes sont des fonctions (membres)

Les méthodes ont toutes les propriétés des fonctions en C++

- Sur-définition (différenciation avec la signature)
- Arguments par défaut
- Définies `inline`
  - avec `inline` en dehors de la classe
  - ou sans le qualificatif `inline` dans la classe
- un objet peut être en argument d'une méthode
  - transmis par valeur
  - transmis par adresse
  - transmis par référence
  - transmis par référence constante
- le retour d'une méthode peut être
  - un objet
  - un pointeur sur un objet
  - une référence sur un objet

# Les méthodes

## Les méthodes sont des fonctions (membres)

Les méthodes ont toutes les propriétés des fonctions en C++

- Sur-définition (différenciation avec la signature)
- Arguments par défaut
- Définies `inline`
  - avec `inline` en dehors de la classe
  - ou sans le qualificatif `inline` dans la classe
- un objet peut être en argument d'une méthode
  - transmis par valeur
  - transmis par adresse
  - transmis par référence
  - transmis par référence constante
- le retour d'une méthode peut être
  - un objet
  - un pointeur sur un objet
  - une référence sur un objet

# Les méthodes

## Les méthodes sont des fonctions (membres)

Les méthodes ont toutes les propriétés des fonctions en C++

- Sur-définition (différenciation avec la signature)
- Arguments par défaut
- Définies `inline`
  - avec `inline` en dehors de la classe
  - ou sans le qualificatif `inline` dans la classe
- un objet peut être en argument d'une méthode
  - transmis par valeur
  - transmis par adresse
  - transmis par référence
  - transmis par référence constante
- le retour d'une méthode peut être
  - un objet
  - un pointeur sur un objet
  - une référence sur un objet

# Les méthodes

## Les méthodes sont des fonctions (membres)

Les méthodes ont toutes les propriétés des fonctions en C++

- Sur-définition (différenciation avec la signature)
- Arguments par défaut
- Définies `inline`
  - avec `inline` en dehors de la classe
  - ou sans le qualificatif `inline` dans la classe
- un objet peut être en argument d'une méthode
  - transmis par valeur
  - transmis par adresse
  - transmis par référence
  - transmis par référence constante
- le retour d'une méthode peut être
  - un objet
  - un pointeur sur un objet
  - une référence sur un objet

# Les méthodes

## Les méthodes sont des fonctions (membres)

Les méthodes ont toutes les propriétés des fonctions en C++

- Sur-définition (différenciation avec la signature)
- Arguments par défaut
- Définies `inline`
  - avec `inline` en dehors de la classe
  - ou sans le qualificatif `inline` dans la classe
- un objet peut être en argument d'une méthode
  - transmis par valeur
  - transmis par adresse
  - transmis par référence
  - transmis par référence constante
- le retour d'une méthode peut être
  - un objet
  - un pointeur sur un objet
  - une référence sur un objet

# Les méthodes

## Les méthodes sont des fonctions (membres)

Les méthodes ont toutes les propriétés des fonctions en C++

- Sur-définition (différenciation avec la signature)
- Arguments par défaut
- Définies `inline`
  - avec `inline` en dehors de la classe
  - ou sans le qualificatif `inline` dans la classe
- un objet peut être en argument d'une méthode
  - transmis par valeur
  - transmis par adresse
  - transmis par référence
  - transmis par référence constante
- le retour d'une méthode peut être
  - un objet
  - un pointeur sur un objet
  - une référence sur un objet

# Les méthodes

## Les méthodes sont des fonctions (membres)

Les méthodes ont toutes les propriétés des fonctions en C++

- Sur-définition (différenciation avec la signature)
- Arguments par défaut
- Définies `inline`
  - avec `inline` en dehors de la classe
  - ou sans le qualificatif `inline` dans la classe
- un objet peut être en argument d'une méthode
  - transmis par valeur
  - transmis par adresse
  - transmis par référence
  - transmis par référence constante
- le retour d'une méthode peut être
  - un objet
  - un pointeur sur un objet
  - une référence sur un objet

# Les méthodes

## Les méthodes sont des fonctions (membres)

Les méthodes ont toutes les propriétés des fonctions en C++

- Sur-définition (différenciation avec la signature)
- Arguments par défaut
- Définies `inline`
  - avec `inline` en dehors de la classe
  - ou sans le qualificatif `inline` dans la classe
- un objet peut être en argument d'une méthode
  - transmis par valeur
  - transmis par adresse
  - transmis par référence
  - transmis par référence constante
- le retour d'une méthode peut être
  - un objet
  - un pointeur sur un objet
  - une référence sur un objet

# Les méthodes

## Les méthodes sont des fonctions (membres)

Les méthodes ont toutes les propriétés des fonctions en C++

- Sur-définition (différenciation avec la signature)
- Arguments par défaut
- Définies `inline`
  - avec `inline` en dehors de la classe
  - ou sans le qualificatif `inline` dans la classe
- un objet peut être en argument d'une méthode
  - transmis par valeur
  - transmis par adresse
  - transmis par référence
  - transmis par référence constante
- le retour d'une méthode peut être
  - un objet
  - un pointeur sur un objet
  - une référence sur un objet

# Autoréférence d'un objet

## Accès à ses propres membres

On peut accéder aux membres de l'objet courant sans le désigner ou en le désignant explicitement.

## Pointeur sur l'objet courant

Pour désigner l'objet courant, on dispose de

- `this` qui est un pointeur sur l'objet courant
- `*this` qui désigne l'objet courant
- pour accéder aux champs de `this`, on utilise l'opérateur `->`

# Autoréférence d'un objet

## Accès à ses propres membres

On peut accéder aux membres de l'objet courant sans le désigner ou en le désignant explicitement.

## Pointeur sur l'objet courant

Pour désigner l'objet courant, on dispose de

- `this` qui est un pointeur sur l'objet courant
- `*this` qui désigne l'objet courant
- pour accéder aux champs de `this`, on utilise l'opérateur `->`

# Autoréférence d'un objet

## point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
  Point();
  Point Somme(Point);
  };
```

## point.cc (C)

```
Point::Point()
{ this->x=0; // c'est x
  y=0; // c'est this->y
}
Point Point::Somme(Point P)
{
return Point(x + P.x , this->y + P.y);
}
```

# Autoréférence d'un objet

## point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
  Point();
  Point Somme(Point);
  };
```

## point.cc (C)

```
Point::Point()
{ this->x=0; // c'est x
  y=0; // c'est this->y
}
Point Point::Somme(Point P)
{
return Point(x + P.x , this->y + P.y);
}
```

Section 11

# Amitiés

- Fonctions amies
- Classes amies

# Fonctions amies

## une fonction amie

est une fonction

- qui sans être membre de la classe a le droit d'accéder aux membres privés (méthodes ou attributs)
- elle est déclarée dans la classe avec le qualificatif `friend`
- elle est définie sans le qualificatif `friend`
- elle ne dispose pas d'un accès à `this`

## Attention

Une fonction amie n'est pas une méthode de la classe

# Fonctions amies

## une fonction amie

est une fonction

- qui sans être membre de la classe a le droit d'accéder aux membres privés (méthodes ou attributs)
- elle est déclarée dans la classe avec le qualificatif `friend`
- elle est définie sans le qualificatif `friend`
- elle ne dispose pas d'un accès à `this`

## Attention

Une fonction amie n'est pas une méthode de la classe

# Fonctions amies : exemple

## Déclaration dans point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
  ...
  friend bool coincide (Point, Point);
  friend void Affiche(Point);
  };
```

## Définition dans point.cc (C)

```
bool coincide(Point P, Point Q)
{
return (P.x==Q.x)&&(P.y==Q.y);
}
void Affiche(Point P)
{
cout<<"("<<P.x<<","<<P.y<<")";
}
```

# Fonctions amies : exemple

## Déclaration dans point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
  ...
  friend bool coincide (Point, Point);
  friend void Affiche(Point);
  };
```

## Définition dans point.cc (C)

```
bool coincide(Point P, Point Q)
{
return (P.x==Q.x)&&(P.y==Q.y);
}
void Affiche(Point P)
{
cout<<"("<<P.x<<","<<P.y<<")";
}
```

# Fonctions amies : exemple

## Appel main.cc (C)

```
int main()
{ Point P(2,3), Q(4,5);
cout<<coincide(P,Q);
Affiche(P);
return 0:
}
```

Appel d'une fonction amie comme une fonction standard car ce n'est pas une méthode

# Fonctions amies : exemple

## Appel main.cc (C)

```
int main()
{ Point P(2,3), Q(4,5);
cout<<coincide(P,Q);
Affiche(P);
return 0:
}
```

Appel d'une fonction amie comme une fonction standard car ce n'est pas une méthode

# Fonctions amies : usage

## Usage

Les fonctions amies sont utilisées :

- dans la cas où elles sont partagées par d'autres classes, la même fonction est définie comme amie des deux classes
- dans le cas où elle ne peut pas être définie comme une méthode

## Surcharge et amitié

Dans le cas de surcharge des opérateurs qui ne peuvent pas être des méthodes, on utilise des fonctions amies. Par exemple pour la surcharge de l'opérateur «

# Fonctions amies : usage

## Usage

Les fonctions amies sont utilisées :

- dans la cas où elles sont partagées par d'autres classes, la même fonction est définie comme amie des deux classes
- dans le cas où elle ne peut pas être définie comme une méthode

## Surcharge et amitié

Dans le cas de surcharge des opérateurs qui ne peuvent pas être des méthodes, on utilise des fonctions amies. Par exemple pour la surcharge de l'opérateur «

# Classes amies

## Classe amie

Une classe amie `A` d'une classe `B` est une classe

- qui a le droit d'accès à tous les membres de `B`
- elle doit être déclarée ou définie dans la classe `B` (la classe qui accorde le droit d'accès)
- elle est précédée du mot réservé `friend`

## Syntaxe

```
class B {
    friend class A;
    // A est une amie de
       B
private:
    int i;
};
```

```
class A {
public:
  //Constructeur de A avec en argument une
      instance de B
   A(B b){ b.i = 0;}
  // la relation d'amitie autorise l'acces aux
      membres de B
};
```

# Classes amies

## Classe amie

Une classe amie `A` d'une classe `B` est une classe

- qui a le droit d'accès à tous les membres de `B`
- elle doit être déclarée ou définie dans la classe `B` (la classe qui accorde le droit d'accès)
- elle est précédée du mot réservé `friend`

## Syntaxe

```
class B {
    friend class A;
    // A est une amie de
        B
private:
    int i;
};
```

```
class A {
public:
  //Constructeur de A avec en argument une
      instance de B
   A(B b){ b.i = 0;}
  // la relation d'amitie autorise l'acces aux
      membres de B
};
```

# Classes amies

## Classe amie : propriétés

- La relation d'amitié n'est pas symétrique : "si A est une amie de B, B n'est pas automatiquement une amie de A"
- La relation d'amitié n'est pas transitive : "les amies de mes amies ne sont pas mes amies"
- La relation d'amitié ne peut être héritée.

# Classes amies

## Classe amie : propriétés

- La relation d'amitié n'est pas symétrique : "si A est une amie de B, B n'est pas automatiquement une amie de A"
- La relation d'amitié n'est pas transitive : "les amies de mes amies ne sont pas mes amies"
- La relation d'amitié ne peut être héritée.

# Classes amies

## Classe amie : propriétés

- La relation d'amitié n'est pas symétrique : "si A est une amie de B, B n'est pas automatiquement une amie de A"
- La relation d'amitié n'est pas transitive : "les amies de mes amies ne sont pas mes amies"
- La relation d'amitié ne peut être héritée.

Section 12

# Agrégation

# Notion d'agrégation

## Une notion de POO

L'agrégation permet de définir une entité comme étant liée à plusieurs entités de classes différentes. C'est une généralisation de la composition, qui n'entraîne pas l'appartenance.

## En C++

C'est une donnée membre (attribut) instance d'une autre classe. Pour `A` et `B` deux classes

```
class A {
private:
  B x;
public:
  ...
  A(int);// constructeur de A
  ...
};
```

# Notion d'agrégation

## Une notion de POO

L'agrégation permet de définir une entité comme étant liée à plusieurs entités de classes différentes. C'est une généralisation de la composition, qui n'entraîne pas l'appartenance.

## En C++

C'est une donnée membre (attribut) instance d'une autre classe. Pour `A` et `B` deux classes

```
class A {
private:
  B x;
public:
  ...
  A(int);// constructeur de A
  ...
};
```

# Agrégation : appel du constructeur

Comment appeler le constructeur de `B` avec l'argument `n` ?

## Solution 1

Avec une liste d'initialisation :

```
A::A(int n):x(n)
{
...
}
```

## Solution 2

Appel du constructeur par défaut de B puis affectation

```
A::A(int n)
 {
  x=n;
  }
```

# Agrégation : exemple

## ObjetGraphique.h (C)

```
class ObjetGraphique {
private:
    Point pointBase;
    int couleur;
    int epaisseur;
public:
    ObjetGraphique(int, int, int, int);
};
```

## ObjetGraphique.cc (C)

```
ObjetGraphique::ObjetGraphique(int x, int y, int coul, int epais) :
    pointBase(x,y), couleur(coul), epaisseur(epais)
{}
```

Section 13

# Vecteur d'instances de classe

# Vecteur d'instances de classe

## Syntaxe

Pour une classe `C`, `C* ident;`

## Sémantique et fonctionnement

Pour pouvoir déclarer un vecteur (tableau) d'instances de la classe `C`, elle doit posséder un constructeur sans paramètre (par défaut) appelé lors de l'allocation mémoire.

# Vecteur d'instances de classe

## Syntaxe

Pour une classe `C`, `C* ident;`

## Sémantique et fonctionnement

Pour pouvoir déclarer un vecteur (tableau) d'instances de la classe `C`, elle doit posséder un constructeur sans paramètre (par défaut) appelé lors de l'allocation mémoire.

# Vecteur d'instances de classe : exemple

## Déclaration dans point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
  Point(){x=0,y=0;}
  ...
  };
```

## main.cc (U)

```
Point * P, segment, triangle;
P = new Point(2,3); // appel du const. avec
    parametres
segment = new Point[2]; // appel du const. par
    defaut 2 fois
triangle = new Point[3]; // appel du const. par
    defaut 3 fois
...
delete P;
delete [] segment;
delete [] triangle;
```

# Vecteur d'instances de classe : exemple

## Déclaration dans point.h (C)

```
class Point{
private :
// attributs
  int x,y;
public :
  Point(){x=0,y=0;}
  ...
  };
```

## main.cc (U)

```
Point * P, segment, triangle;
P = new Point(2,3); // appel du const. avec
    parametres
segment = new Point[2]; // appel du const. par
    defaut 2 fois
triangle = new Point[3]; // appel du const. par
    defaut 3 fois
...
delete P;
delete [] segment;
delete [] triangle;
```