IFT339
Structures de données

Thème 3 : Mécanisme d'abstraction en C++

Aïda Ouangraoua
Département d'informatique

UNIVERSITÉ DE
SHERBROOKE

Rappel

- **Classe** : permet construction de nouveaux types abstraits
- **Caractérisques de la POO** : encapsulation, héritage, polymorphisme
- **Classe en C++** : Encapsulation de la représentation d'un type et des opérations permettant de manipuler les éléments du type (Accès aux attributs et opérations : public, protected ou private)

```
class NomClasse
{
    private:
        //representation
        <Type> attribut1;
        <Type> attribut2;
   public:
        //operations
        <SpecificationFonction1>;
        <SpecificationFonction2>;
}
```

Rappel

- **Classe** : permet construction de nouveaux types abstraits
- **Caractérisques de la POO** : encapsulation, héritage, polymorphisme
- **Classe en C++** : Encapsulation de la représentation d'un type et des opérations permettant de manipuler les éléments du type (Accès aux attributs et opérations : public, protected ou private)

```
class NomClasse
{
    private:
        //representation
        <Type> attribut1;
        <Type> attribut2;
   public:
        //operations
        <SpecificationFonction1>;
        <SpecificationFonction2>;
}
```

```
class Point
{
    private:
        float abcisse;
        float ordonnee;
   public:
        Point(); // constructeur
        Point(float, float); // constructeur
        float calculerDistance(Point);
}
```

Rappel

- **Instance d'une classe** : un élément du type, obtenu en appelant un constructeur Exemple: `Point p1 ,p2(1,1), p3(2,2);`

Rappel

- **Instance d'une classe** : un élément du type, obtenu en appelant un constructeur Exemple: `Point p1 ,p2(1,1), p3(2,2);`

- La définition d'une classe peut être dans un seul fichier, ou répartie dans plusieurs fichiers :
  - **fichier header (.h)** : spécification de la représentation et des opérations du type
  - **un ou plusieurs fichiers cpp (.cpp)** : implémentation des opérations spécifiées. Dans ce cas :
    - Le fichier .h doît être inclus dans le fichier .cpp (Exemple: `#include "point.h"`)
    - L'opérateur de portée «::» est utilisé pour préciser la classe à laquelle appartiennent les opérations

Rappel

- **Instance d'une classe** : un élément du type, obtenu en appelant un constructeur Exemple: `Point p1 ,p2(1,1), p3(2,2);`

- La définition d'une classe peut être dans un seul fichier, ou répartie dans plusieurs fichiers :
  - **fichier header (.h)** : spécification de la représentation et des opérations du type
  - **un ou plusieurs fichiers cpp (.cpp)** : implémentation des opérations spécifiées. Dans ce cas :
    - Le fichier .h doît être inclus dans le fichier .cpp (Exemple: `#include "point.h"`)
    - L'opérateur de portée «::» est utilisé pour préciser la classe à laquelle appartiennent les opérations

  Exemple:

```
float Point::calculerDistance(Point p){
    float difference_abscisse = abcisse - p.abcisse;
    float difference_ordonnee = ordonnee - p.ordonnee;
    return sqrt((diff_abs*diff_abs) + (diff_ord*diff_ord));
}
```

Classe abstraite, concrète, de base, dérivée

- **Classe abstraite** : classe que l'on ne peut pas instancier (impossible de faire appel à un constructeur) car elle contient au moins une méthode virtuelle pure (méthode virtuelle sans implémentation)

```
virtual <SpecificationFonction1> = 0 ;
```

- **Classe concrète** : classe que l'on peut instancier

Classe abstraite, concrète, de base, dérivée

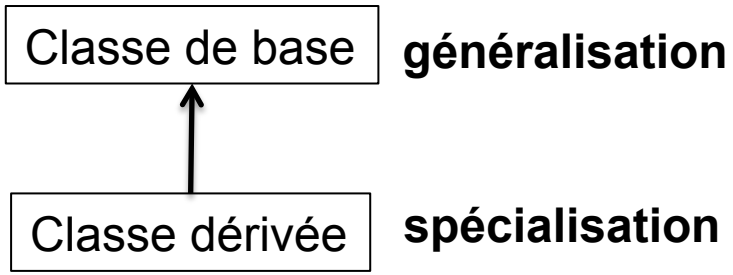
- **Classe abstraite** : classe que l'on ne peut pas instancier (impossible de faire appel à un constructeur) car elle contient au moins une méthode virtuelle pure (méthode virtuelle sans implémentation)

```
virtual <SpecificationFonction1> = 0 ;
```

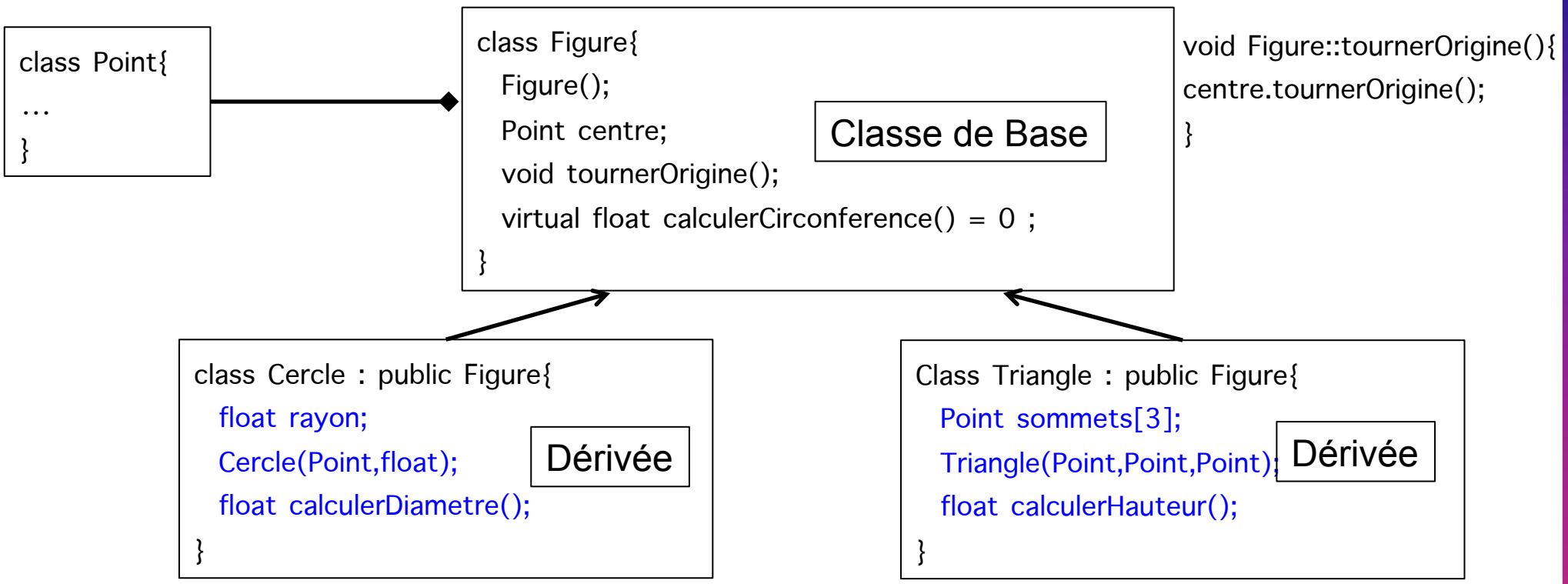
- **Classe concrète** : classe que l'on peut instancier

- **Classe de base** : classe dont est dérivée une autre classe par héritage
- **Classe dérivée** : classe héritant d'une autre autre classe (de base)
  - peut ajouter ou redéfinir des caractéristiques.
- **Héritage** : toutes les caractéristiques de la classe de base sont données à la classe dérivée (Type d'héritage: public, protected ou private)

```
class NomClasseDerivee : public NomClasseBase
{
//Ajout de caracteristiques (attrbuts, operations)
specifiques a la classe derivee
}
```



Exemple



Ajout

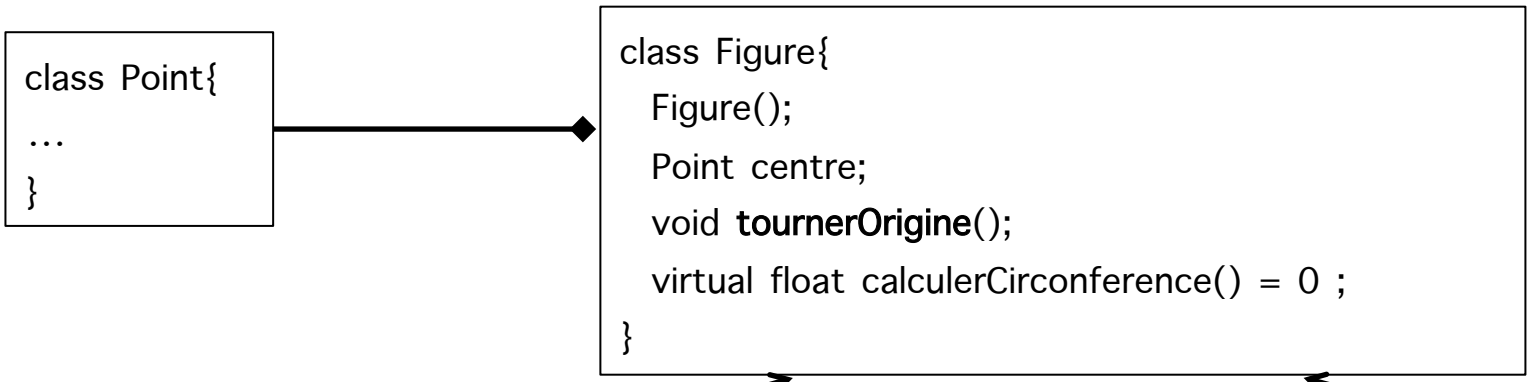
Définition de méthode virtuelle pure héritée

Re-définition de méthode

```
float Cercle::calculerCirconference(){
float circonference;
circonference = PI * pow(rayon,2);
return circonference;}

float Cercle::calculerDiametre(){
float diametre;
diametre = 2 * rayon;
return diametre;}
```

```
float Triangle::calculerCirconference(){
float circonference = … ;
return circonference;}

void Triangle::tournerOrigine(){
…}

float Triangle::calculerHauteur(){
float hauteur = …;
return hauteur;}
```

Exemple (Illustration du Polymorphisme)



Ajout

Définition de Méthode virtuelle pure héritée

Re-définition de méthode

```
class Cercle : public Figure{
  float rayon;
  Cercle(Point,float);
  float calculerDiametre();
}
```

```
float Cercle::calculerCirconference(){…}
```

```
Class Triangle : public Figure{
  Point sommets[3];
  Triangle(Point,Point,Point);
  float calculerHauteur();
}
```

```
float Triangle::calculerCirconference(){…}
void Triangle::tournerOrigine(){…}
```

```
Figure liste_figures[2] = {Cercle(Point(0,0), 2),
Triangle(Point(0,0), Point(1,0), Point(0,1))};

Liste_figure[0].tournerOrigine();                    //Figure::tournerOrigine()
float circ1 = Liste_figure[0].calculerCirconference();   //Cercle::calculerCirconference()

Liste_figure[1].tournerOrigine();                    //Triangle::tournerOrigine()
float circ2 = Liste_figure[1].calculerCirconference();   //Triangle::calculerCirconference()
```

Classe générique

- **Classe générique** : classe servant de modèle pour la création de plusieurs classes identiques, à quelques types de caractéristiques près.

```
templale <typename NomType>
class NomClasse
{
    NomType attribut;
    ...
}
```

Classe générique

- **Classe générique** : classe servant de modèle pour la création de plusieurs classes identiques, à quelques types de caractéristiques près.

```
templale <typename NomType>
class NomClasse
{
   NomType attribut;
   ...
}
```

Exemple:

```
templale <typename TYPE>
class Point{
   TYPE abcisse; // abscisse de type TYPE
   TYPE ordonnee;

   Point(); // constructeur
   Point(TYPE, TYPE); // constructeur
   TYPE calculerAbscisse();
   float calculerDistance(Point<TYPE>);
}
```

Classe générique

- **Classe générique** : classe servant de modèle pour la création de plusieurs classes identiques, à quelques types de caractéristiques près.

```
templale <typename NomType>
class NomClasse
{
    NomType attribut;
    ...
}
```

Exemple:

```
templale <typename TYPE>
class Point{
    TYPE abcisse; // abscisse de type TYPE
    TYPE ordonnee;

    Point(); // constructeur
    Point(TYPE, TYPE); // constructeur
    TYPE calculerAbscisse();
    float calculerDistance(Point<TYPE>);
}
```

```
template <typename TYPE>
TYPE Point<TYPE>::calculerAbscisse(){
    return abscisse;
}

template <typename TYPE>
float Point<TYPE>::calculerDistance(Point<TYPE> p){
    return sqrt(pow(abscisse – p.abcisse,2) + pow(ordonnee – p.ordonnee,2));
}
```

Classe générique

- **Classe générique** : classe servant de modèle pour la création de plusieurs classes identiques, à quelques types de caractéristiques près.

```
templale <typename NomType>
class NomClasse
{
   NomType attribut;
   ...
}
```

```
Point<int> p1(0,0), p2(1,1);
Point<float> p3(0.0,0.0), p4(0.0,0.0);
float dist1 = p1.calculerDistance(p2);
float dist2 = p3.calculerDistance(p4);
```

Exemple:

```
templale <typename TYPE>
class Point{
   TYPE abcisse; // abscisse de type TYPE
   TYPE ordonnee;

   Point(); // constructeur
   Point(TYPE, TYPE); // constructeur
   TYPE calculerAbscisse();
   float calculerDistance(Point<TYPE>);
}
```

Classe générique

- **Classe générique** : classe servant de modèle pour la création de plusieurs classes identiques, à quelques types de caractéristiques près.

```
templale <typename NomType>
class NomClasse
{
   NomType attribut;
   ...
}
```

Exemple:

```
templale <typename TYPE>
class Point{
   TYPE abcisse; // abscisse de type TYPE
   TYPE ordonnee;

   Point(); // constructeur
   Point(TYPE, TYPE); // constructeur
   TYPE calculerAbscisse();
   float calculerDistance(Point<TYPE>);
}
```

```
Point<int> p1(0,0), p2(1,1);
Point<float> p3(0.0,0.0), p4(0.0,0.0);
float dist1 = p1.calculerDistance(p2);
float dist2 = p3.calculerDistance(p4);
```

Polymorphisme

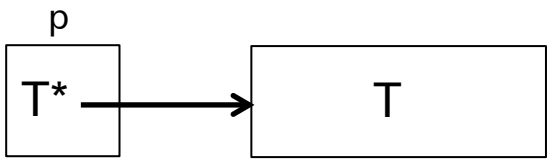
Point<int>::calculerDistance

Point<float>::calculerDistance

Pointeur

- **Pointeur** : variable dont la valeur est l'adresse d'une autre variable

```
T* p ; //declaration de p, pointeur vers une variable de type T
```



Exemple:

```
int*  pI; //pI pointe vers un int
Point* pP; // pP pointe vers un Point
float* pF; //pF pointe vers un float
```

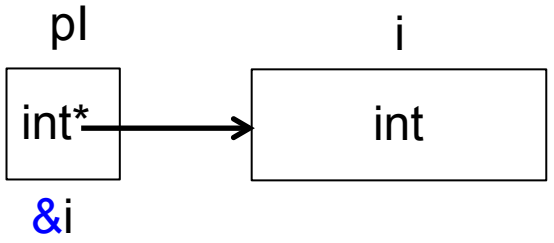
Pointeur

❑ **Opérateur d'indirection &** : adresse d'une variable donnée

&var; //Si var est de type T, alors &var contient l'adresse de var et est de type T*

Exemple:

```
int i = 0; // i est un int
int* pI; //pI doit pointer vers un int
pI  = &i; // pI contient l'adresse de i
```

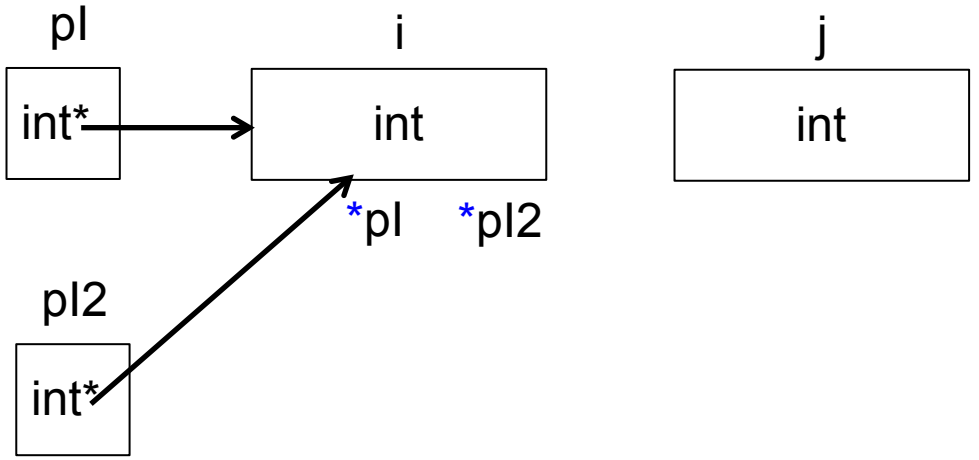


❑ **Opérateur de déréférencement *** : valeur d'une variable pointée

p; //Si p est de type T, alors *p contient la valeur de la variable vers laquelle p pointe, et est de type T

Exemple:

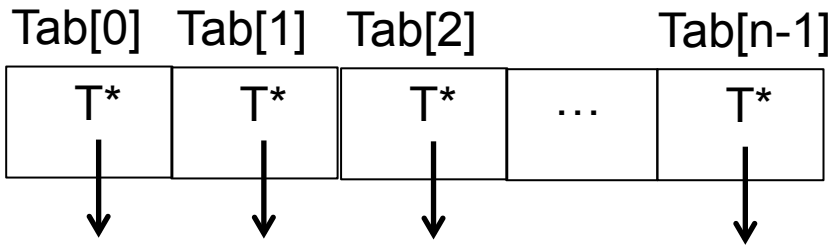
```
int i = 0; // i est un int
int* pI = &i; //pI pointe vers un i
int j = *pI ;// j = 0 (valeur de i)
*pI = 1; // i prend la valeur 1
int* pI2 = pI;
*pI2 = 2; // i prend la valeur 2
```



Pointeur

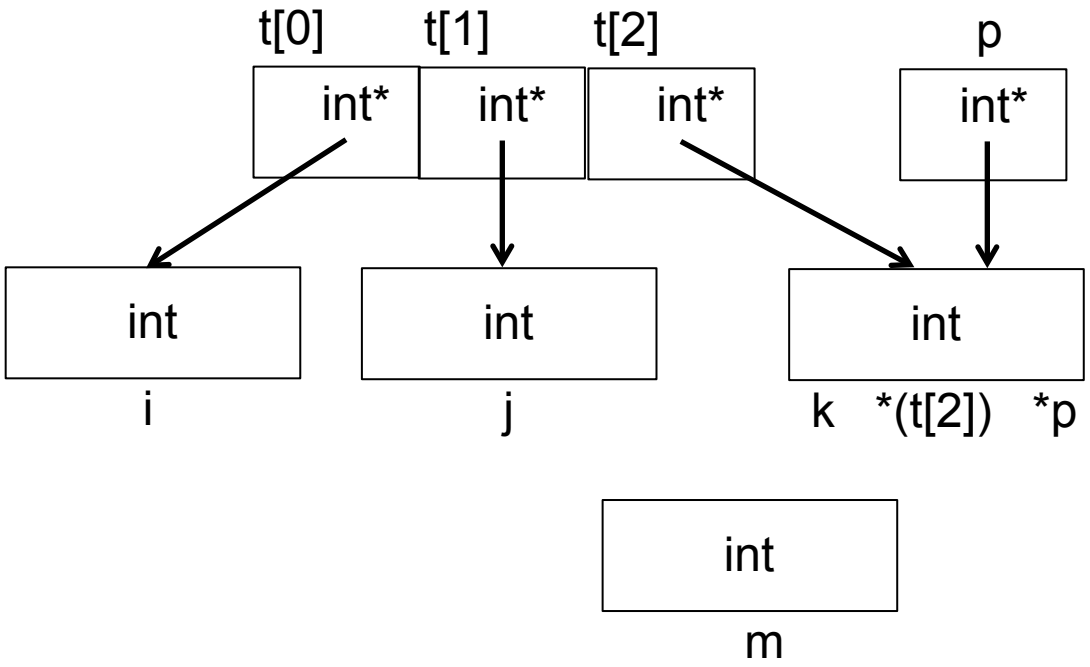
❑ Tableau de pointeur

T* Tab[n] ; //declaration de nomTab, tableau de n pointeurs vers des variables de type T



Exemple:

```
int i(0), j(2), k(4); // i,j,k sont des int
int* t[3]; // t est un tableau de 3 int*
t[0] = &i; // t[2] est initialise a &i
t[1] = &j; // t[2] est initialise a &j
t[2] = &k; // t[2] est initialise a &k
int m = *t[2]; // m == 4 (valeur de k)
int* p =  t[2]; // p pointe vers k
*t[2] = 5; // k prend la valeur 5
*p = 6; // k prend la valeur 6
// m a garde la valeur 4
```



Pointeur: Placement de l’attribut const (immuable)

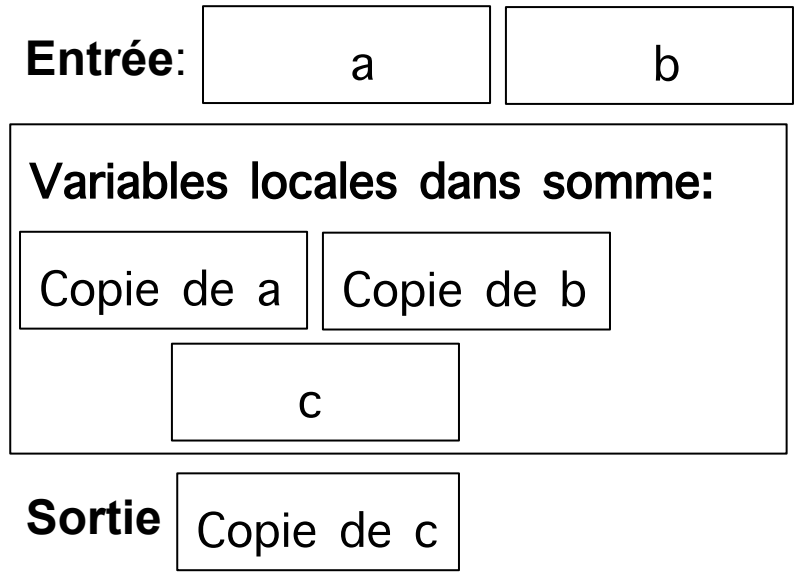
Syntaxe	**Pointeur pmodifiable**	**Objet pointé (*p) modifiable**
int* p;	oui	oui
const int* p;	oui	non
int* const p;	non	oui
const int* const p;	non	non

Opérations d'une classe (type T) = fonctions

- **Passage de paramètres et retour de fonction** :
  - Par valeur ou par référence
  - Constant ou modifiable

- Par valeur : une variable locale qui est une copie de la variable en entrée ou en sortie est utilisée

```
int somme(int a, int b){
    int c = a + b;
    return c;
}
```



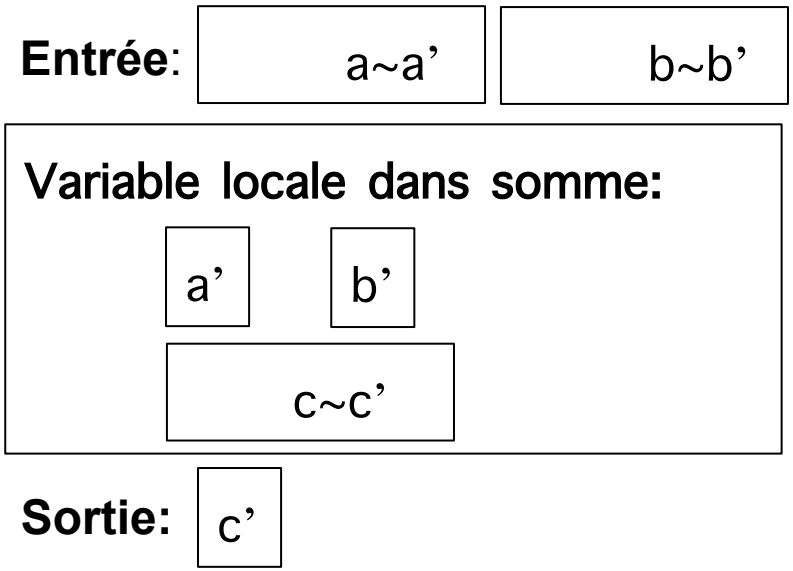
  - Avantage : pas de modification de a et b, dont les valeurs restent inchangées après l'exécution de la fonction « somme »
  - Limite : copie de variables qui peuvent être de grande taille

Opérations d'une classe (type T) = fonctions

- **Passage de paramètres et retour de fonction** :
  - Par valeur ou par référence
  - Constant ou modifiable

- Par référence : un identificateur synonyme de la variable paramètre est utilisée

```
int & somme(int & a, int & b){
    int c = a + b;
    return c;
}
```



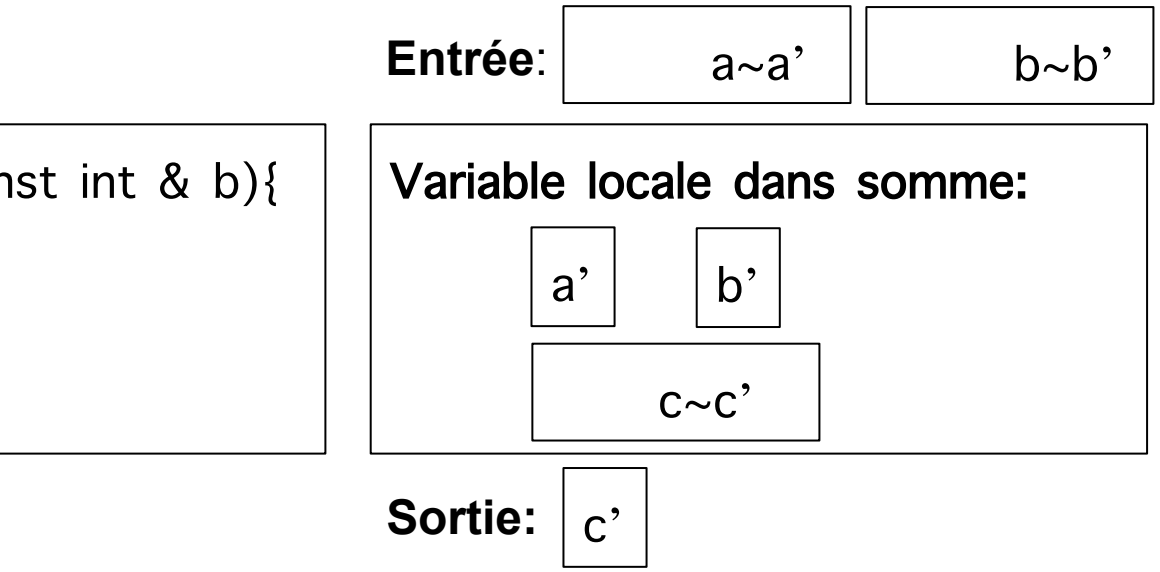
  - Avantage : Pas de copie dans des variables locales
  - Limite : les valeurs de a et b peuvent être modifiées dans « somme »

Opérations d'une classe (type T) = fonctions

- **Passage de paramètres et retour de fonction** :
  - Par valeur ou par référence
  - Constant ou modifiable

- Constant : le mot-clé « const » permet de rendre une variable immuable

```
const int & somme(const int & a, const int & b){
    int c = a + b;
    return c;
}
```



  - Avantage : Les valeurs de a~a' et b~b' ne peuvent être modifiées dans « somme » ; La valeur de c ne sera pas modifiable après la sortie de la fonction « somme ».

Opérations de base d'une classe (type T)

- **Constructeur :** pour initialiser une instance de T

```
Point(0,0);
```

- **Destructeur :** pour nettoyer la mémoire allouée pour une instance
- **Affectateur** : pour donner une nouvelle valeur à une instance de T

```
int a(1); b(2), c;
c = a + b;
```

- **Convertisseurs :** pour convertir d'un autre type au type T ou inversément
- **Sélecteurs (« get »):** pour accéder aux valeurs de la représentation d'une instance (bonne pratique de les définir privés ou protégés)
- **Mutateurs (« set »)**: pour modifier les valeurs de la représentation d'une instance (bonne pratique de les définir privés ou protégés)

Paramètre implicite « this » passé par référence

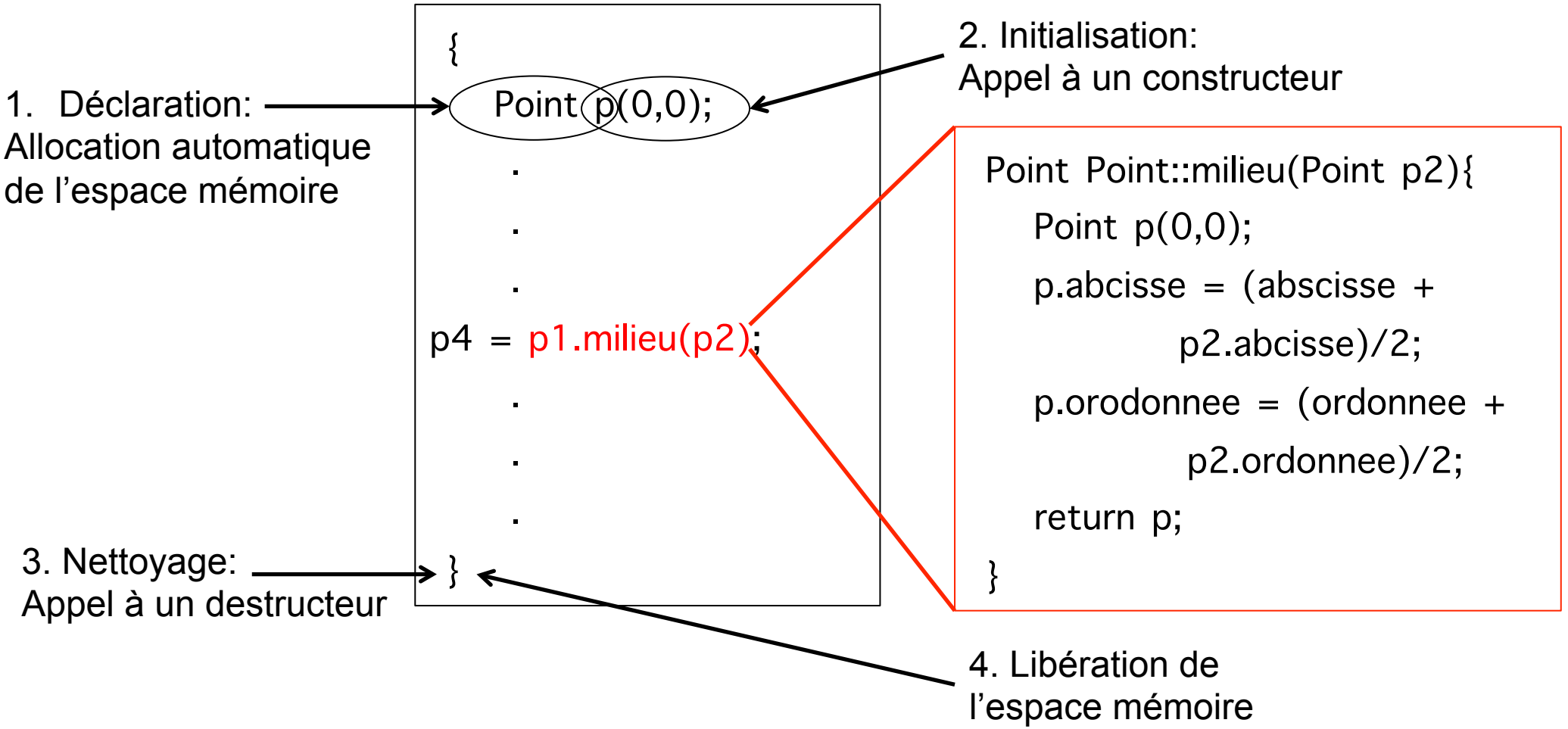
- Toutes les opérations définies dans une classe ont un paramètre implicite « this » passé par référence (objet appelant la fonction)

- Placement de l'attribut « const » pour le paramètre implicite « this »:

```
float Point::distance(const Point) const;
```

Constructeur, Destructeur

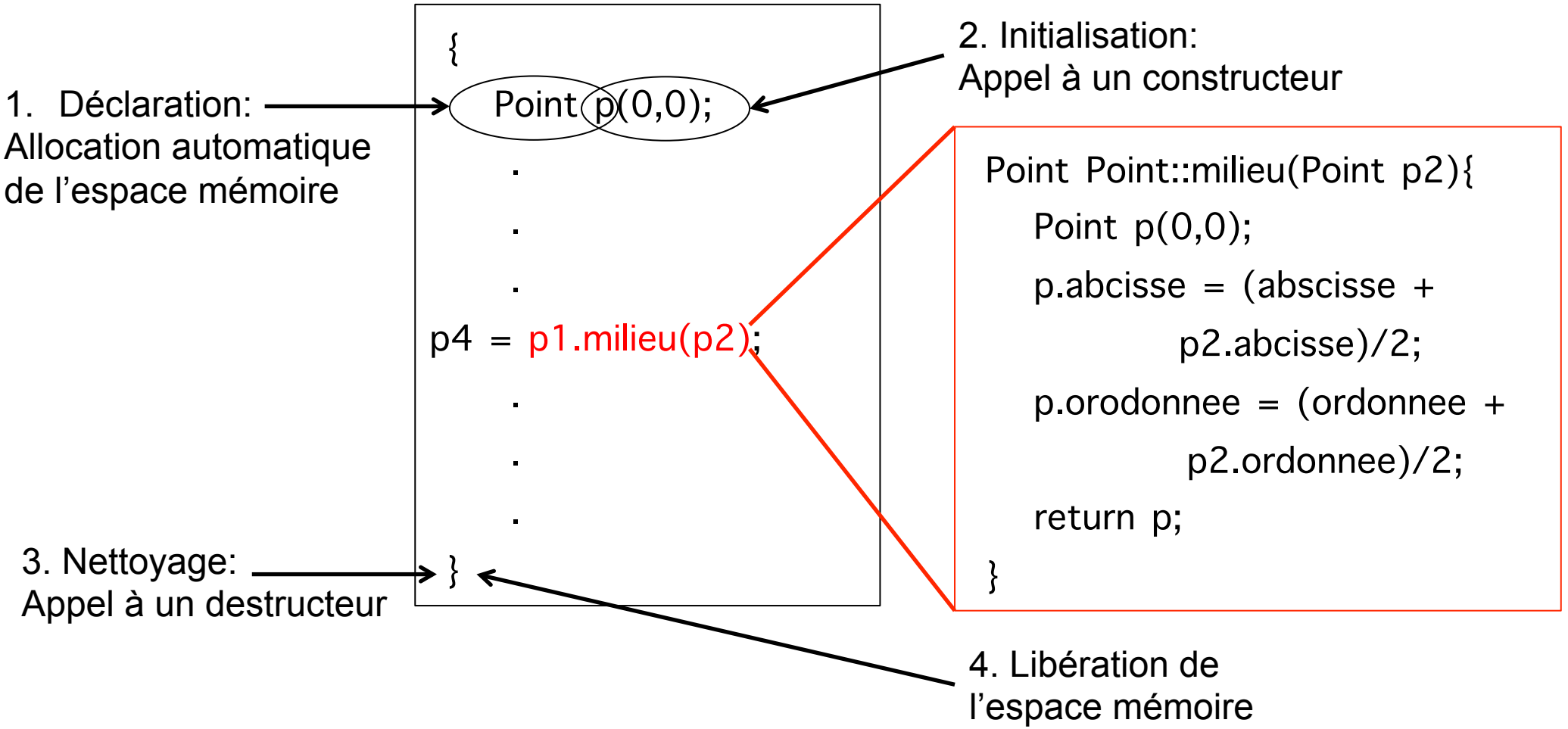
❑ 4 événements particuliers dans la vie d'un objet



- 1. et 4. peuvent être gérés automatiquement par le système qui connaît la définition générale de l'objet
- Par contre, 2. et 3. doivent être réalisés par l'objet lui-même, qui connaît plus spécifiquement son propre comportement

Constructeur, Destructeur

❑ 4 événements particuliers dans la vie d'un objet



- La durée de vie de l'objet va de 1. à 4.
- La portée de l'objet (parties où il est accessible et lisible) va de 2. à 3, avec des trous correspondant aux parties où il n'est pas accessible. Exemple : p n'est pas accessible dans « milieu ».

Constructeur

- Permet d'initialiser un objet (nécessaire pour initialiser correctement les objets de types complexes)
- Porte le nom de la classe
- Déclaré dans la portée de la
- classe
- A un paramètre implicite : this (objet créé, passé par référence)

```
class Point{
    float abcisse;
    float ordonnee;

    Point(); // constructeur sans parametre
    Point(float, float); // constructeur avec
                         // parametres
    Point(const Point &) // constructeur
                         // par copie
}
```

❑ Plusieurs constructeurs possibles:

- Constructeur sans paramère explicite : généré automatiquement par le compilateur, si non défini (résultat correct ?)
- Constructeur avec paramètres explicites : un ou plusieurs
- Constructeur par copie : prend en paramètre un objet de même type pour initialiser la représentation de l'objet créé (aussi généré automatiquement, mais avec une copie superficielle)

Constructeur

- Peut être défini à l'extérieur de la classe, mais toujours dans la portée de la classe

```
Point::Point{
    abcisse = ordonnee = 0.0;
}
Point::Point(float abs,float ord){
    abcisse = abs;
    ordonnee = ord;
}
Point::Point(const Point & p) {
    abcisse = p.abscisse;
    ordonnee = p.ordonnee;
}
```

```
// Exemple de delegation de construction
// Le constructeur sans parametre appelle
// le constructeur avec parametre ➔
// Reutilisabilite
Point::Point() : Point(0.0,0.0){
}
```

- Délégation de construction : un constructeur peut en appeler un autre

Destructeur

- Permet de nettoyer un objet (nécessaire pour nettoyer correctement les objets pour lesquels de la mémoire est allouée dynamiquement)
- nom : ~nomClasse
- Déclaré et défini dans la portée de la classe
- A un seul paramètre passé par référence : this (implicite)

```
class Point{
    float abcisse;
    float ordonnee;

    ~Point(); // destructeur
}
```

```
Point::~Point(){
// Rien a faire car aucune memoire
// n'est allouee dynamiquement
}
```

Affectateur

- Surcharge de l'opérateur =, permettant d'affecter à l 'objet appelant, la valeur d'un autre objet

```
Point p(0,0),p1(1,1),p2(2,2);
p = p1.milieu(p2);
```

- Différent du constructeur par copie car il ne permet pas de créer l'objet. Il le met à jour uniquement.
- Généré automatiquement par le compilateur, si il n'est pas défini, mais avec une copie superficielle (comme pour le constructeur par copie)

```
class Point{
    float abcisse;
    float ordonnee;

    void operator=(const Point &); //
}
```

```
void Point::operator=(const Point & p){
    abcisse = p.abscisse;
    ordonnee = p. ordonnee;
}
```

Gestion des opérateurs générés automatiquement et appelés directement par le compilateur (constructeur, destructeur, affectateur)

- Mot-clé « default » : indique au compilateur d'utiliser la version par défaut

- Mot-clé « delete » : indique au compilateur de supprimer la version par défaut

```
class Point{
    float abcisse;
    float ordonnee;

    Point() = default;
    Point(const Point &) = delete;
    ~Point() = default;
}
```

Convertisseurs (pour une classe T)

- Pour convertir d'un autre type A au type T : définir un constructeur dans T n'ayant qu'un seul paramètre explicite de type A.

```
class Point{
    Point(float);   // convertisseur de float vers Point
}
```

- Pour convertir du type T à un autre type A : définir une surcharge de operator A

```
class Point{
    void operator float(); // convertisseur de Point vers float
}
```

Autres opérations d'une classe (type T)

- **Surcharge d'opérateurs** : utilisation du mot-clé « operator »

```
class Point{
    …
    void operator=(const Point &); // affectateur
    bool operator==(const Point &); // comparaison: egalite
    bool operator!=(const Point &); // comparaison: difference
    Point operator-(); // operateur unaire : oppose
    Point operator-(const Point &); //operateur binaire: soustraction
    Point operator+(const Point &); //operateur binaire: somme
    void operator--(); //operateur unaire: decrementation
    void operator++(); //operateur unaire: incrementation
    …
}
```

- Le nombre de paramètres d'un opérateur ne peut être modifié
- Une bonne pratique est que la surcharge d'un opérateur corresponde à une extension naturelle de sa sémantique initiale, mais ce n'est pas obligatoire.

Autres opérations d'une classe (type T)

- **Fonctions amies** : utilisation du mot-clé « friend »

  - Ne font pas partie de la classe, mais ont accès aux parties privées et protégées de la classe
  - N'ont pas le paramètre implicite « this »
  - Déclarées dans la portée de la classe

```
class Point{
    ...
    friend float distance(Point , Point);
}
```

```
friend float Point::distance(Point p1, Point p2)){
    float dist;
    dist = sqrt(pow(p1.abscisse -p2.abscisse, 2)
            +pow(p1.ordonnee -p2.ordonnee, 2));
    return dist;
}
```

Autres opérations d'une classe (type T)

- **Membres statiques** : utilisation du mot-clé « static »

  - Attribut ou fonction en une seule copie à laquelle toutes les instances de la classe ont accès
    - Exemple : attribut pour compter le nombre d'instances d'une classe
    - Une fonction statique n'a pas de paramètre implicite « this » (similaire à une fonction amie)

```
class Point{
    static int nbPoints;
    Point();
    ~Point();
    static float aire(Point ,Point, Point);
}
```

```
Point::nbPoints = 0;
Point::Point (){
    abscisse = ordonnee = 0;
    nbPoints += 1;
}
Point::~Point (){
    abscisse = ordonnee = 0;
    nbPoints -= 1;
}
```