La programmation orientée objet (POO) en C++
Deuxième partie

Thierry Vaira

BTS SN Option IR

v.1.2 - 10 octobre 2018

Sommaire

Les attributs statiques

- Un membre donnée déclaré avec l'attribut `static` est **partagé par tous les objets de la même classe**. C'est un attribut de classe et non d'objet.
- Il existe même lorsque aucun objet de cette classe n'a été créé.
- Un membre donnée statique doit être initialisé explicitement, à l'extérieur de la classe (même s'il est privé), en utilisant l'opérateur de résolution de portée (::) pour spécifier sa classe. Il y a une exception pour un **membre statique constant**. Dans ce cas, l'initialisation se fait dans la classe elle-même.
- En général, son initialisation se fait dans le fichier `.cpp` de définition de la classe.

Les méthodes statiques

- Lorsqu'une fonction membre a **une action indépendante d'un quelconque objet de sa classe**, on peut la déclarer avec l'attribut `static`.
- Dans ce cas, une telle fonction peut être appelée, sans mentionner d'objet particulier, en préfixant simplement son nom du nom de la classe concernée, suivi de l'opérateur de résolution de portée (::).
- Il faut tenir compte :
  - Les méthodes statiques ne peuvent pas accéder aux attributs de la classe car il est possible qu'aucun objet de cette classe n'ait été créé.
  - Les méthodes statiques peuvent accéder aux membres données statiques car ceux-ci existent même lorsque aucun objet de cette classe n'a été créé.

Déclaration des membres statiques

- Les membre statiques sont souvent utilisés pour compter le nombre d'instances créées.

```
class Lampe
{
   private:
      ...
      static int nbLampes; // je suis un membre donnée statique

   public:
      ...
      static int getNbLampes(); // je suis une méthode statique

};
```

Définition des membres statiques

```
// Initialisation d'un membre statique (dans le fichier .cpp)
int Lampe::nbLampes = 0;

Lampe::Lampe() // Constructeur
{
   ...
   nbLampes++; // un objet Lampe de plus !
}

Lampe::~Lampe() // Destructeur
{
   nbLampes--; // un objet Lampe de moins !
}

// Je retourne le nombre d'objets Lampe existants à un instant donné
int Point::getNbLampes()
{
   return nbLampes;
}
```

Utilisation des membres statiques

```
cout << "Nb lampes = " << Lampe::getNbLampes() << endl; // Affiche 0

Lampe l1;
Lampe *l2 = new Lampe(100);
Lampe l3(100);

cout << "Nb lampes = " << Lampe::getNbLampes() << endl; // Affiche 3

delete l2;

cout << "Nb lampes = " << Lampe::getNbLampes() << endl; // Affiche 2
```

Liste d'initialisation

- Pour initialiser un objet lors de l'appel d'un constructeur, C++ propose une syntaxe spéciale pour cela, la **liste de l'initialisation du constructeur**
- La liste d'initialisation permet d'utiliser le constructeur de chaque donnée membre, et ainsi d'éviter une affectation après coup
- La liste d'initialisation doit être utilisée pour certains cas :
  - les références
  - les attributs constants
  - l'héritage (que l'on verra plus tard)

```
Date::Date(int j, int m, int annee) : jour(j), mois(m), annee(annee)
{
}
```

Attributs constants

Il faut distinguer :

- les **constantes de classe** : on préfixe l'**attribut statique** avec le mot clé `const`. Il est initialisé dans la déclaration de la classe. Sa valeur est constante pour tous les objets de cette classe.
- les **constantes d'objet** : on préfixe l'**attribut** avec le mot clé `const`. Chaque objet pourra initialiser son attribut constant dans la liste d'initialisation du constructeur.

```
class A
{
   private:
      const static int MAX = 10; // une constante de classe
      const int n; // une constante d'objet
   // ...
};

A::A() : n(100) { ... }
```

Rôle d'un constructeur de copie

Le **constructeur de copie** est appelé dans :

- la création d'un objet à partir d'un autre objet pris comme modèle
- le passage en paramètre d'un objet par valeur à une fonction ou une méthode
- le retour d'une fonction ou une méthode renvoyant un objet

```
Date d1; // Appel du constructeur par défaut

Date d2 = d1; // Appel du constructeur de copie pour instancier d2
Date d2(d1); // Appel du constructeur de copie pour instancier d2

Date d3; // Appel du constructeur par défaut

d3 = d1; // Appel de l'opérateur = pour affecter d1 à d3
```

Toute autre duplication (au cours de la vie d'un objet) sera faite par l'opérateur d'affectation (=).

Déclaration d'un constructeur de copie

La déclaration d'un constructeur de copie est la suivante :

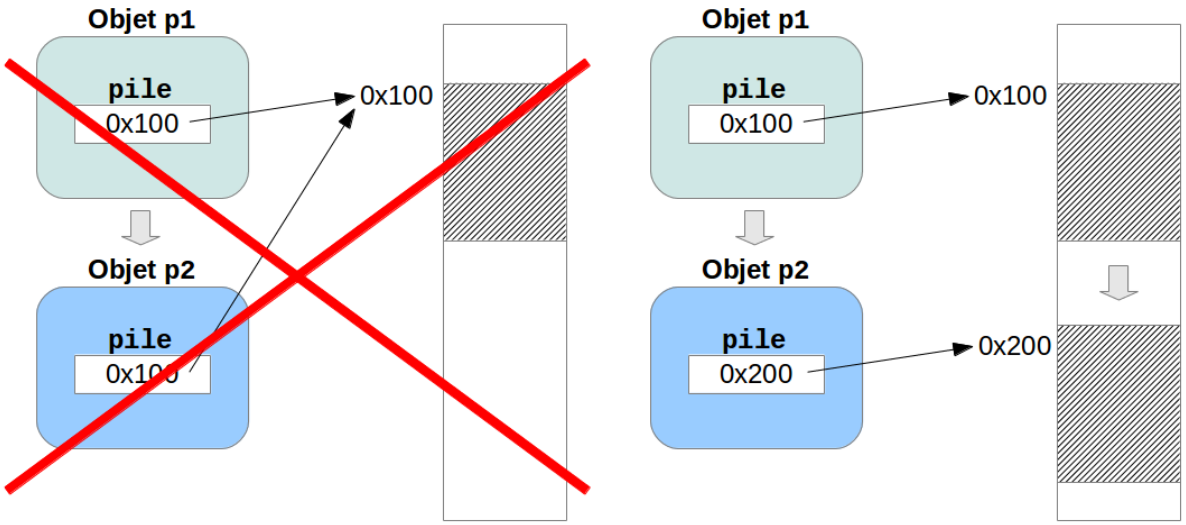
```
class T
{
   public:
      T(const T&);
};
```

Donc pour une classe `Date` :

```
Date::Date(const Date &d)
{  // On recopie les attributs un par un
   jour = d.jour;
   mois = d.mois;
   annee = d.annee;
}
```

Dangers

Il faut faire très attention avec les classes qui manipulent de la mémoire dynamique et des pointeurs :




```
class PileChar
{
   private:
      char *pile;
      unsigned int sommet;
      unsigned int max;
};
```

Exemple

Il faut faire très attention avec les classes qui manipulent de la mémoire dynamique et des pointeurs :

```
PileChar::PileChar(const PileChar &p) : max(p.max), sommet(p.sommet)
{
   // on alloue dynamiquement le nouveau tableau de caractères
   pile = new char[max];

   unsigned int i;

   // on recopie les éléments de la pile
   for (i = 0; i < sommet ; i++)
      pile[i] = p.pile[i];
}
```

Rôle de l'opérateur d'affectation

- L'**opérateur d'affectation** (=) est un **opérateur de copie d'un objet vers un autre**
- L'objet affecté est déjà créé **sinon** c'est le **constructeur de copie** qui sera appelé

```
Date d1; // Appel du constructeur par défaut

Date d2 = d1; // Appel du constructeur de copie pour instancier d2

// Différent de :

Date d3; // Appel du constructeur par défaut

d3 = d1; // Appel de l'opérateur = pour affecter d1 à d3
```

Déclaration de l'opérateur d'affectation

La déclaration de l'opérateur d'affectation est la suivante :

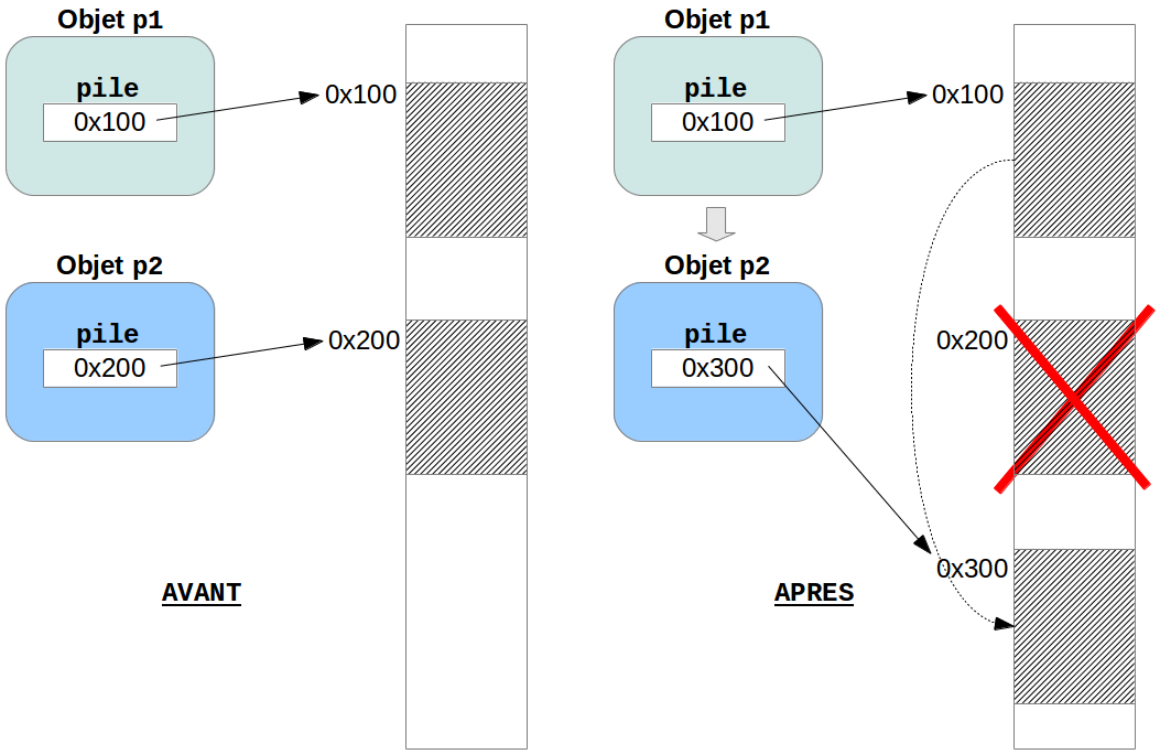
```
class T
{
   public:
      T& operator=(const T&);
};
```

Donc pour une classe Date :

```
Date& Date::operator = (const Date &d)
{ // vérifions si on ne s'auto-copie pas !
   if (this != &d) { // Alors on recopie les attributs un par un
      jour = d.jour;
      mois = d.mois;
      annee = d.annee;
   }
   return *this;
}
```

Dangers

➠ Il faut faire très attention avec les classes qui manipulent de la mémoire dynamique et des pointeurs :



➠ D'autre part, cet opérateur renvoie une **référence** sur `T` afin de pouvoir l'utiliser avec d'autres affectations. En effet, l'opérateur d'affectation est associatif à droite : `a=b=c` est évaluée comme `a=(b=c)`. Ainsi, la valeur renvoyée par une affectation doit être à son tour modifiable.

Exemple

Il faut faire très attention avec les classes qui manipulent de la mémoire dynamique et des pointeurs :

```
PileChar& PileChar::operator = (const PileChar &p)
{
   // vérifions si on ne s'auto-copie pas !
   if (this != &p)
   {
      delete [] pile; // on libère l'ancienne pile

      max = p.max;
      sommet = p.sommet;
      pile = new char[max]; // on alloue une nouvelle pile
      unsigned int i;
      for (i = 0; i < sommet ; i++) pile[i] = p.pile[i]; // on recopie les
           éléments de la pile
   }

   return *this;
}
```

Bonnes pratiques

Une classe T est dite sous **forme canonique** (ou forme normale ou forme standard) de **Coplien** si elle présente les méthodes suivantes :

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

Rôle de l'amitié

- L'**unité de protection** en C++ est la **classe**. Ceci implique :
  - Une fonction membre peut accéder à tous les membres de n'importe quel objet de sa classe
  - Par contre, le principe d'encapsulation interdit à une fonction membre d'une classe d'accéder à des données privées d'une autre classe
- L'**amitié** en C++ permet à des fonctions (ou des méthodes) **d'accéder à des membres privées d'une autre classe**
- Les **fonctions amies** se déclarent en faisant précéder la déclaration classique de la fonction du mot clé `friend` à l'intérieur de la déclaration de la classe cible

Fonction ou Méthode amie

```
class A {
    private:
        int a; // Un attribut privé
    public:
        A(int a=0) : a(a) {}
        friend void modifierA(int i); // Une fonction amie
        friend void B::modifierA(int i); // Une méthode amie
};

void modifierA(int i) // Je suis une fonction (amie)
{
    A unA; // un objet de type A

    unA.a = i; // possible car je suis un ami de la classe A
} ...
```

➠ Il est possible de déclarer amie une méthode d'une autre classe, en précisant son nom complet à l'aide de l'opérateur de résolution de portée (::).

Classe amie

```
class A
{
    ...
    friend class Amie; // La classe Amie est une amie : elle pourra accéder à mes
          membres privés
};

class Amie
{
    public:
      void afficherA(const A& unA) // un objet de type A en paramètre
      {
         cout << unA.a << endl; // possible car je suis une amie de la classe A
      }
};
```

Remarques

➠ On remarquera plusieurs choses importantes :

- **l'amitié n'est pas transitive** : **les amis des amis ne sont pas des amis**. Une classe `A` amie d'une classe `B`, elle-même amie d'une classe `C`, n'est pas amie de la classe `C` par défaut.
- **l'amitié n'est pas héritée** : **mes amis ne sont pas les amis de mes enfants**. Si une classe `A` est amie d'une classe `B` et que la classe `C` est une classe fille de la classe `B`, alors `A` n'est pas amie de la classe `C` par défaut.
- Attention, l'utilisation des classes amies peut aussi traduire un **défaut de conception** (classes amies vs classes dérivées).

Rôle des fonctions inline

➠ C++ présente une **amélioration** des **macros** du langage C avec les **fonctions `inline`** :

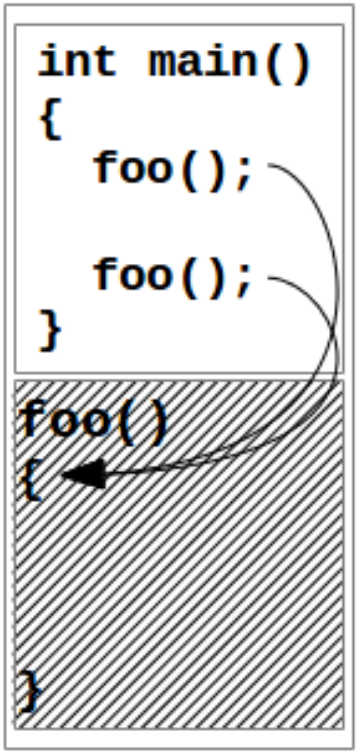
- Elles ont le comportement des fonctions (vérification des arguments et de la valeur de retour)
- Elles sont substituées dans le code après vérification

➠ De manière générale, les fonctions `inline` (ou les macros) ont les caractéristiques suivantes :

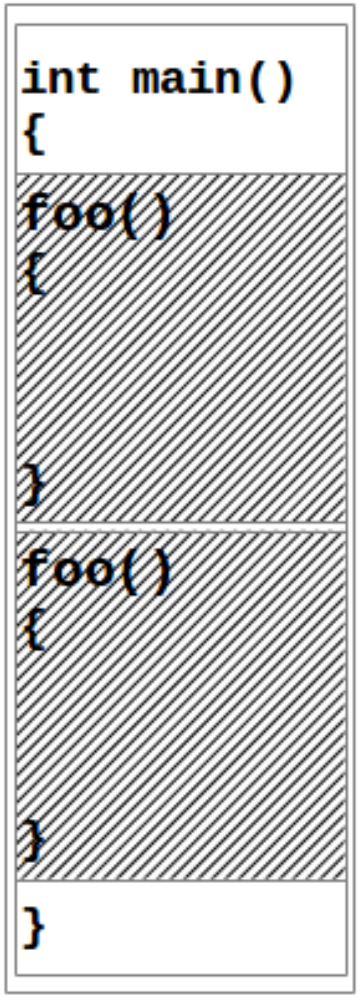
- Avantage : plus rapide qu'une fonction (sauf si la taille du programme devient trop importante)
- Inconvénients : comme le code est généré pour chaque appel, le programme binaire produit est plus volumineux. D'autre part, cela complique la compilation séparée.

Principe

foo( ) est une **fonction**



foo( ) est une **fonction inline (macro)**



Les méthodes *inline*

➠ Il existe deux techniques pour implémenter une méthode `inline` :

```
class Entier
{
    private:
      int a;
    public:
        // 1. lorsque le corps de la méthode est définie directement dans la
             déclaration
        int getEntier() const { return a; } // getEntier() est alors inline
        void setEntier(int i);
};

// 2. lorsqu'on définit une méthode, on ajoute au début de la définition le mot-
     clé inline
inline void Entier::setEntier(int i) // setEntier() est alors inline
{
    a = i;
}
```

Intérêt de la surcharge d'opérateurs

- La **surcharge d'opérateur** permet aux opérateurs du C++ d'avoir une signification spécifique quand ils sont appliqués à des types spécifiques.
- Parmi les nombreux exemples que l'on pourrait citer :
  - `myString + yourString` pourrait servir à concaténer deux objets `string`
  - `maDate++` pourrait servir à incrémenter un objet `Date`
  - `a * b` pourrait servir à multiplier deux objets `Nombre`
  - `e[i]` pourrait donner accès à un élément contenu dans un objet `Ensemble`

Les surcharges habituelles

➠ Les opérateurs C++ que l'on surcharge habituellement :

- Affectation, affectation avec opération (`=`, `+=`, `*=`, etc.) : **Méthode**
- Opérateur « fonction » `()` : **Méthode**
- Opérateur « indirection » `*` : **Méthode**
- Opérateur « crochets » `[]` : **Méthode**
- Incrémentation `++`, décrémentation `--` : **Méthode**
- Opérateur « flèche » et « flèche appel » `->` et `->*` : **Méthode**
- Opérateurs de décalage « et » : **Méthode**
- Opérateurs `new` et `delete` : **Méthode**
- Opérateurs de lecture et écriture sur flux « et » : **Fonction**
- Opérateurs dyadiques genre « arithmétique » (`+`,`-`,`/` etc) : **Fonction**

➠ Les autres opérateurs ne peuvent pas soit être surchargés soit il est déconseillé de le faire.

Technique n°1 : méthode

- La **première technique** pour surcharger les opérateurs consiste à les considérer comme des **méthodes** normales de la classe sur laquelle ils s'appliquent.

```
// A Op B se traduit par A.operatorOp(B)

// t1 == t2; // équivalent à : t1.operator==(t2)
// t1 += t2; // équivalent à : t1.operator+=(t2)
// ...

bool Date::operator == (const Date &d)
{
   if(jour != d.jour || mois != d.mois || annee != d.annee)
      return false;

   return true;
}
```

Technique n°2 : fonction amie (1/2)

- La **deuxième technique** utilise la surcharge d'opérateurs externes sous forme de **fonctions amies**.
- La définition de l'opérateur ne se fait plus dans la classe qui l'utilise, mais en dehors de celle-ci. Dans ce cas, tous les opérandes de l'opérateur devront être passés en paramètres.

```
// A Op B se traduit par operatorOp(A, B)

// t1 + t2; // équivalent à : operator+(t1, t2)

class T
{
   public:
      friend T operator+(const T &a, const T &b);
};
```

Technique n°2 : fonction amie (2/2)

```
Nombre operator+(const Nombre &n1, const Nombre &n2) {
   // solution n° 1 :
   Nombre result = n1;
   result.valeur += n2.valeur;
   return result;
   // solution n° 2 : si l'opérateur += a été surchargé
   Nombre result = n1;
   return result += n2;
}
```

L'avantage de cette syntaxe est que l'opérateur est réellement **symétrique**, contrairement à ce qui se passe pour les opérateurs définis à l'intérieur de la classe.

```
Nombre n1, n2, n3; // Des objets de type Nombre
n3 = n1 + n2; // Appel de l'opérateur + puis de l'opérateur de copie (=)
n3 = n2 + n1; // idem car l'opérateur est symétrique
```

Les opérateurs d'incrémentation et de décrémentation (1/2)

➠ Le problème :

- Les opérateurs d'incrémentation (++) et de décrémentation −− ont la même notation mais représentent deux opérateurs en réalité.
- En effet, ils n'ont pas la même signification, selon qu'ils sont placés avant (préfixés) ou après (suffixés) leur opérande.
- Ne possédant pas de paramètres (ils ne travaillent que sur l'objet), il est donc **impossible de les différencier par surcharge**.

➠ La **solution** qui a été adoptée est de les différencier en donnant un **paramètre fictif de type `int`** à l'un d'entre eux.

- opérateurs préfixés : ++ et −− ne prennent pas de paramètre et doivent renvoyer une référence sur l'objet lui-même
- opérateurs suffixés : ++ et −− prennent un paramètre `int` fictif (que l'on n'utilisera pas) et peuvent se contenter de renvoyer la valeur de l'objet

Les opérateurs d'incrémentation et de décrémentation (2/2)

Exemple pour l'opérateur d'incrémentation ++ :

```
// Opérateur préfixe : incrémente puis retourne une référence sur l'objet
T& T::operator++(void)
{
   ++x ; // incrémente la variable

   return *this ; // se retourne lui-même
}

// Opérateur suffixe : retourne d'abord l'objet et puis l'incrémente
T T::operator++(int n)
{
   // crée un objet temporaire
   T tmp(x); // peut nuire gravement aux performances !
   ++x;

   return tmp;
}
```

Notions de flux (flot)

- Un **flot** est un **canal recevant (flot d'« entrée ») ou fournissant (flot de « sortie ») de l'information**.
- Ce canal est associé à un **périphérique** ou à un **fichier**. *Remarque : sous GNU/Linux, les périphériques sont des fichiers.*
- Un **flot d'entrée** est un objet de type `istream` tandis qu'un **flot de sortie** est un objet de type `ostream`.
- Le flot `cout` est un **flot de sortie** prédéfini connecté à la sortie standard `stdout` d'un programme (l'écran par défaut).
- De même, le flot `cin` est un **flot d'entrée** prédéfini connecté à l'entrée standard `stdin` d'un programme (le clavier par défaut).
- Il existe aussi la sortie standard des erreurs `stderr` utilisable avec `cerr` (redirigée vers l'écran par défaut).

Surcharge des opérateurs de flux

- On surchargera les opérateurs de flux « et » pour une classe quelconque, sous forme de **fonctions amies** :

```
ostream & operator << (ostream & sortie, const type_classe & objet1)
{
   // Envoi sur le flot sortie des membres de objet en utilisant
   // les possibilités classiques de << pour les types de base
   // c'est-à-dire des instructions de la forme :
   // sortie << ..... ;
   return sortie ;
}

istream & operator >> (istream & entree, type_de_base & objet)
{
   // Lecture des informations correspondant aux différents membres de objet
   // en utilisant les possibilités classiques de >> pour les types de base
   // c'est-à-dire des instructions de la forme :
   // entree >> ..... ;
   return entree ;
}
```

Surcharge de l'opérateur de flux «

- Si on implémente la surcharge de l'opérateur de flux de sortie « pour qu'il affiche un objet `Point` de la manière suivante : `<x,y>`, on pourra alors écrire :

```
ostream & operator << (ostream & os, const Point & p)
{
   os << "<" << p.x << "," << p.y << ">";
   return os;
}

// Exemple :
Point p0, p1(4, 0.0), p2(2.5, 2.5);

cout << "P0 = " << p0 << endl; // Affiche P0 = <0,0>
cout << "P1 = " << p1 << endl;
cout << "P2 = " << p2 << endl;
```

Surcharge de l'opérateur de flux »

- Si on implémente la surcharge de l'opérateur de flux d'entrée » pour qu'il lise un objet `Point` de la manière suivante : `<x,y>`, on pourra alors écrire :

```
istream & operator >> (istream & is, Point & p) {
   char c; double x, y;
   if ((c = is.get()) == '<') is >> x;
   else  is.clear(ios::failbit);
   if ((c = is.get()) == ',') is >> y;
   else  is.clear(ios::failbit);
   if (is.eof() || ((c = is.get()) != '>')) is.clear(ios::failbit);
   if (is.eof()) is.clear(ios::failbit);
   if (is) { // saisie valide ?
      p.x = x; p.y = y; }
   return is;
}

// Exemple :
Point p0;
cin >> p0; // saisie : <2.5,1.5>
```