Cours C++ : Initiation POO (première partie)

Sommaire

La programmation orientée objet

Principe

La **programmation orientée objet (POO)** consiste à **définir des objets logiciels et à les faire interagir entre eux**.

Notion de classe

Une **classe déclare des propriétés communes** à un ensemble d'objets.

Une classe représentera donc une **catégorie d'objets**.

Elle apparaît comme un **type** ou un *moule* à partir duquel il sera possible de créer des objets.

```
class Point
{
   double x, y; // nous sommes des propriétés de la classe Point
};
```

Une classe Point

Notion d'objets

Concrètement, un **objet** est une **structure de données** (**ses attributs** c.-à-d. des variables) qui définit son **état** et une **interface** (**ses méthodes** c.-à-d. des fonctions) qui définit son **comportement**.

Un objet est créé à partir d'un **modèle** appelé **classe**. Chaque objet créé à partir de cette classe est une **instance** de la classe en question.

Un objet posséde une **identité** qui permet de distinguer un objet d'un autre objet (son nom, une adresse mémoire).

```
Point point; // je suis une instance (un objet) de la classe Point
```

Un objet point

Notion de visibilité

Le C++ permet de préciser le **type d'accès des membres** (**attributs** et **méthodes**) d'un objet. Cette opération s'effectue au sein des classes de ces objets :

- **public** : les membres publics peuvent être utilisés dans et par n'importe quelle partie du programme.
- **privé** (*private*) : les membres privés d'une classe ne sont accessibles que par les objets de cette classe et non par ceux d'une autre classe.

```
class Point
{
   private:
      double x, y; // nous sommes des membres privés de la classe Point

   public:
      string nom; // je suis un membre public de la classe Point
};
```



```
Point point; // je suis une instance (un objet) de la classe Point

point.x = 0.; // erreur d'accès : 'double Point::x' is private
point.y = 0.; // idem
point.nom = "O"; // pas d'erreur car nom est déclaré public
```

Les accès private et public

Il existe aussi un accès **protégé** (protected) en C++ que l'on abordera avec la notion d'héritage.

Notion d'encapsulation

L'encapsulation est l'idée de **protéger les variables contenues dans un objet et de ne proposer que des méthodes pour les manipuler**.

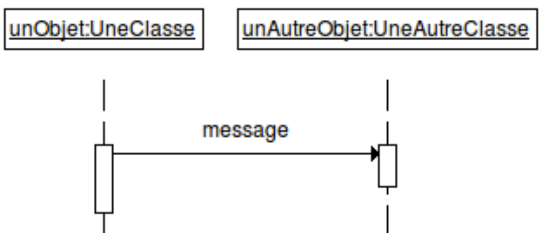
En respectant ce principe, toutes les variables (attributs) d'une classe seront donc privées.

L'objet est ainsi vu de l'extérieur comme une "boîte noire" possédant certaines propriétés et ayant un comportement spécifié. C'est le comportement d'un objet qui modifiera son état.

Notion de messages

Un objet est une structure de données encapsulées qui répond à un **ensemble de messages**. Cette **structure de données (ses attributs) définit son état** tandis que l'**ensemble des messages (ses méthodes) décrit son comportement**.

L'ensemble des messages forme ce que l'on appelle l'**interface de l'objet**. Les objets interagissent entre eux en **s'échangeant des messages**.



La réponse à la réception d'un message par un objet est appelée **une méthode**. Une **méthode est donc la mise en oeuvre du message** : elle décrit la réponse qui doit être donnée au message.

```
class Point
{
   public:
      void afficher(); // je suis une méthode publique de la classe Point
};

Point point; // je suis une instance (un objet) de la classe Point

// Réception du message "afficher" :
point.afficher(); // provoque l'appel de la méthode afficher() de la classe Point
```

Réception d'un message

Notion de surdéfinition (ou surcharge)

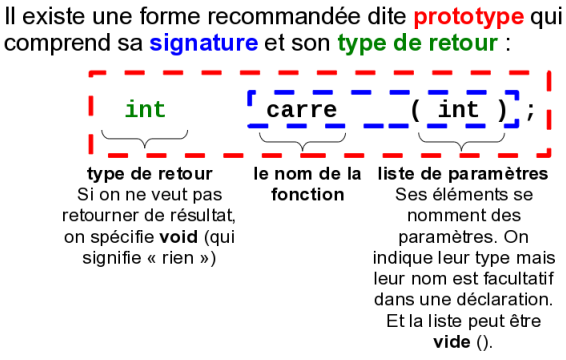
Il est généralement conseillé d'attribuer des noms distincts à des fonctions différentes. Cependant, lorsque des fonctions effectuent la même tache sur des objets de type différent, il peut être pratique de leur attribuer des noms identiques. L'utilisation d'un même nom pour des fonctions (ou méthodes) s'appliquant à des types différents est nommée **surcharge**.

Ce n'est pas possible de réaliser cela en C mais seulement en C++.

```
// Un seul nom au lieu de print_int, print_float, ...
void print(int);
void print(float);

int main (int argc, char **argv)
{
   int n = 2;
   float x = 2.5;

   print(n); // produit -> je suis print et j'affiche un int : 2
   print(x); // produit -> je suis print et j'affiche un float : 2.5

   return 0;
}

void print(int a)
{
   cout << "je suis print et j'affiche un int : " << a << endl;
}

void print(float a)
{
   cout << "je suis print et j'affiche un float : " << a << endl;
}
```

Surcharge de fonctions

Une **surdéfinition (ou surcharge)** permettra donc **d'utiliser plusieurs méthodes qui portent le même nom au sein d'une même classe avec une *signature différente***.



Le type de retour ne fait pas partie de la signature d'une fonction (ou d'une méthode). La surcharge ne peut donc s'appliquer que sur le type et/ou le nombre des paramètres.

Création de classes et d'objets

Construction d'objets

Pour **créer des objets à partir d'une classe**, il faut ... **un constructeur** :

- Un constructeur est chargé d'**initialiser un objet de la classe**.
- Il est appelé **automatiquement au moment de la création** de l'objet.
- Un constructeur est une **méthode qui porte toujours le même nom que la classe**.
- Il existe quelques contraintes :
  - Il peut avoir des paramètres, et des valeurs par défaut.
  - Il peut y avoir plusieurs constructeurs pour une même classe.
  - Il n'a jamais de type de retour.

On le **déclare** de la manière suivante :

```
class Point
{
   private:
      double x, y; // nous sommes des attributs de la classe Point

   public:
      Point(double x, double y); // je suis le constructeur de la classe Point
};
```

Point.h

Le code C++ ci-dessus correspond à la déclaration de la classe `Point`. Il doit être écrit dans un fichier en-tête (*header*) `Point.h`.

Il faut maintenant **définir** ce constructeur afin qu'il **initialise tous les attributs de l'objet au moment de sa création** :

```
// Je suis le constructeur de la classe Point
Point::Point(double x, double y)
{
   // je dois initialiser TOUS les attributs de la classe
   this->x = x; // on affecte l'argument x à l'attribut x
   this->y = y; // on affecte l'argument y à l'attribut y
}
```

Point.cpp

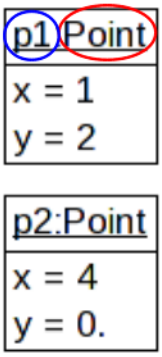
On doit faire précéder chaque méthode de `Point::` pour préciser au compilateur que ce sont des membres de la classe `Point`.

Le code C++ ci-dessus correspond à la définition du constructeur la classe `Point`. Toutes les définitions doivent être placées dans un fichier `C++ Point.cpp`.

Le mot clé "**this**" permet de désigner l'adresse de l'objet sur laquelle la fonction membre a été appelée. On peut parler d"'auto-pointeur" car l'objet s'auto-désigne (sans connaître son propre nom). Ici, cela permet de différencier les deux variables x (et y).

On pourra alors **créer nos propres points** :

UML	C++

	`Point p1(1, 2);`
	
	`Point p2(4, 0.);`

Les objets p1 et p1 sont des instances de la classe Point. Un objet possède sa propre existence et un état qui lui est spécifique (c.-à-d. les valeurs de ses attributs).

Constructeur par défaut

Si vous essayer de créer un objet sans lui fournir une abscisse x et une ordonnée y, vous obtiendrez le message d'erreur suivant :

```
erreur: no matching function for call to Point::Point()'
```

Ce type de constructeur se nomme un **constructeur par défaut**. Son rôle est de créer une instance non initialisée quand aucun autre constructeur fourni n'est applicable.

Le constructeur par défaut de la classe Point sera :

```
Point::Point() // Sans aucun paramètre !
{
   x = 0.;
   y = 0.;
}
```

Paramètre par défaut

Le langage C++ offre la possibilité d'avoir des **valeurs par défaut pour les paramètres d'une fonction (ou d'une méthode)**, qui peuvent alors être sous-entendus au moment de l'appel.

Cette possibilité, permet d'écrire qu'un seul constructeur profitant du mécanisme de valeur par défaut :

```
// Déclaration d'un constructeur (par défaut) de la classe Point
Point(double x=0., double y=0.); // utilisation des paramètres par défaut

// Définition
Point::Point(double x/*=0.*/, double y/*=0.*/)
{
   this->x = x;
   this->y = y;
}
```

```
// Appel du constructeur par défaut de la classe Point (utilise les valeurs par défaut pour
    x et y)
Point p0; // x=0 et y=0

// Appel du constructeur de la classe Point
Point p1(1 , 2); // x=1 et y=2
Point p2(4 , 0.); // x=4 et y=0

// Ici, on utilise la valeur par défaut de y
Point p3(5); // x=5 et y=0
```

Liste d'initialisation

Un meilleur moyen d'affecter des valeurs aux données membres de la classe lors de la construction est la **liste d'initialisation**.

On va utiliser cette technique pour définir le constructeur de la classe `Point` :

```
Point::Point(double x/*=0.*/, double y/*=0.*/) : x(x), y(y)
{
}
```

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 objets qui ne peuvent pas être contruits par défaut : c'est le cas des références et des objets constants.

Allocation dynamique d'objet

Pour allouer dynamiquement un objet en C++, on utilisera l'opérateur `new`. Celui-ci renvoyant une adresse où est créé l'objet en question, il faudra un pointeur pour la conserver. Manipuler ce pointeur, reviendra à manipuler l'objet alloué dynamiquement.

Pour libérer la mémoire allouée dynamiquement en C++, on utilisera l'opérateur `delete`.

```
Point *p3; // je suis pointeur (non initialisé) sur un objet de type Point

p3 = new Point(2,2); // j'alloue dynamiquement un objet de type Point

cout << "p3 = ";
p3->affiche(); // Comme pointC est une adresse, je dois utiliser l'opérateur -> pour accéder
     aux membres de cet objet
//p3->setY(0); // je modifie la valeur de l'attribut y de p3

cout << "p3 = ";
(*p3).affiche(); // cette écriture est possible : je pointe l'objet puis j'appelle sa
    méthode affiche()

delete p3; // ne pas oublier de libérer la mémoire allouée pour cet objet
```

Tableau d'objets

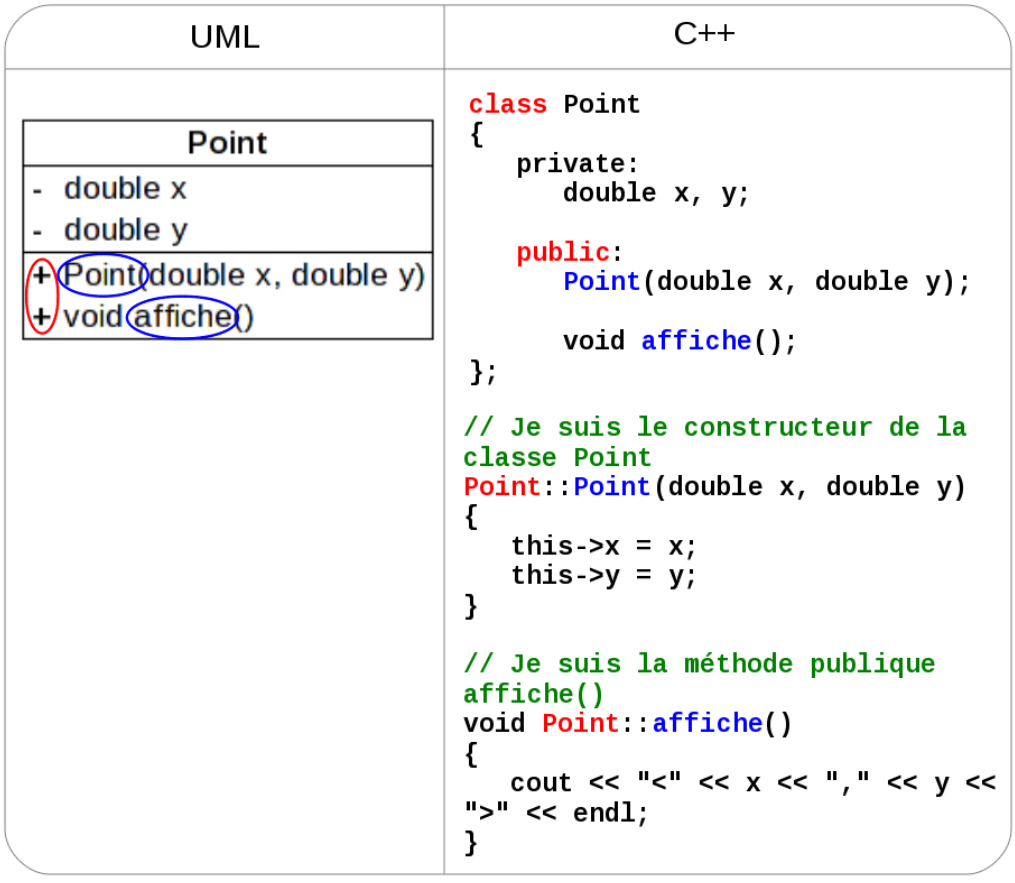
Il est possible de conserver et de manipuler des objets `Point` dans un tableau.

```
// typiquement : les cases d'un tableau de Point
Point tableauDe10Points[10]; // le constructeur par défaut est appelé 10 fois (pour chaque
    objet Point du tableau) !
int i;

cout << "Un tableau de 10 Point : " << endl;
for(i = 0; i < 10; i++)
{
   cout << "P" << i << " = "; tableauDe10Points[i].affiche();
}
cout << endl;
```

Utilisation d'objets

Les services rendus par une classe

Un point pourra s'**afficher**. On aura donc une **méthode** `affiche()` qui produira un affichage de ce type : `<x,y>`



A l'intérieur de la méthode `affiche()` de la classe `Point`, on peut accéder directement à l'abscisse du point en utilisant la donnée membre `x`. De la même manière, on pourra accéder directement à l'ordonnée du point en utilisant la donnée membre `y`.

On utilisera cette méthode dès que l'on voudra **afficher** les coordonnées d'un point :

```
cout << "P1 = ";
p1.affiche();

cout << "P2 = ";
p2.affiche();
```

Ce qui donnera à l'exécution :

```
P1 = <1,2>
P2 = <4,0>
```

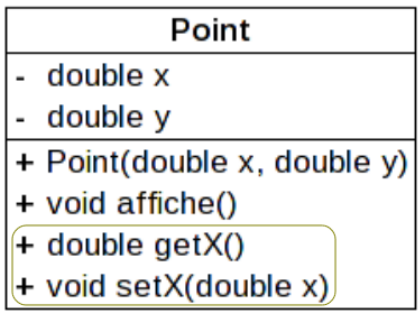
Une méthode publique est un service rendu à l'utilisateur de l'objet.

Les accès contrôlés aux membres d'une classe

Toutes les variables de la classe `Point` étant **privées par respect du principe d'encapsulation**, on veut néanmoins pouvoir connaître son abscisse et pouvoir modifier cette dernière.

Il faut donc créer deux **méthodes publiques** pour **accéder** à l'**attribut** `x` :

UML



C++

```
class Point
{
   private:
      double x, y;

   public:
      Point(double x, double y);
      void affiche();
      double getX() const;
      void setX(double x);

};

// L'accesseur get du membre x
double Point::getX() const
{
   return x;
}

// Le manipulateur set du membre x
void Point::setX(double x)
{
   this->x = x;
}
```

La méthode `getX()` est déclarée constante (`const`). Une méthode constante est tout simplement une méthode qui ne modifie aucun des attributs de l'objet. Il est conseillé de qualifier `const` toute fonction qui peut l'être car cela garantit qu'on ne pourra appliquer des méthodes constantes que sur un objet constant.

La méthode publique `getX()` est un accesseur (*get*) et `setX()` est un manipulateur (*set*) de l'attribut `x`. L'utilisateur de cet objet ne pourra pas lire ou modifier directement une propriété sans passer par un accesseur ou un manipulateur.

On utilisera ces méthodes pour accéder en lecture ou écriture aux membres privés d'un point :

```
// on peut modifier le membre x de P1
p1.setX(5);
// on peut accéder au membre x de P1
cout << "L'abscisse de P1 est " << p1.getX() << endl;

// on peut accéder au membre x de P2
cout << "L'abscisse de P2 est " << p2.getX() << endl;
```

Ce qui donnera à l'exécution :

```
L'abscisse de P1 est 5
L'abscisse de P2 est 4
```

Les objets constants

Les règles suivantes s'appliquent aux objets constants :
- On déclare un objet constant avec le modificateur `const`
- On ne peut appliquer que des méthodes constantes sur un objet constant
- Un objet passé en paramètre sous forme de référence constante est considéré comme constant

Une **méthode constante** est tout simplement une méthode qui ne modifie aucun des attributs de l'objet.

```
class Point
{
   private:
      double x, y;

   public:
      Point(double x=0, double y=0) : x(x), y(y) {}

      double getX() const; // une méthode constante
      double getY() const; // une méthode constante
      void setX(double x);
      void setY(double y);

      void affiche();
};

// L'accesseur get du membre x
double Point::getX() const
{
   return x;
}

// Le manipulateur set du membre x
void Point::setX(double x)
{
   this->x = x;
}
```

```
// etc ...

// La méthode publique affiche()
void Point::affiche()
{
   cout << "<" << x << "," << y << ">" << endl;
}
```

On va tester ces méthodes avec un objet constant :

```
const Point p5(7, 8); // un objet constant

cout << "p5 = ";
p5.affiche(); // la méthode affiche() doit être déclarée const [1]
cout << "L'abscisse de p5 est " << p5.getX() << endl; // pas d'erreur car getX() est
    déclarée const
cout << "L'ordonnée de p5 est " << p5.getY() << endl; // pas d'erreur car getY() est
    déclarée const
p5.setX(5); // vous ne pouvez pas appeler cette méthode car elle n'est pas const [2]
```

On obtient deux erreurs à la compilation ([1] et [2]) :

```
erreur: passing 'const Point' as 'this' argument of 'void Point::affiche()' discards
    qualifiers
erreur: passing 'const Point' as 'this' argument of 'void Point::setX(double)' discards
    qualifiers
```

L'erreur [1] doit être corrigée en déclarant `const` la méthode `affiche()` car elle ne modifie pas les attributs de la classe.

L'erreur [2] est une protection qui nous alerte que l'on ne peut pas modifier un objet constant. Il faut donc supprimer cette ligne ou retirer le caractère `const` de l'objet `p5`.

Il est donc conseillé de qualifier `const` toute fonction qui peut l'être car cela garantit qu'on ne pourra appliquer des méthodes constantes que sur un objet constant.

Passage de paramètre par référence

Rappel : En C++, il est possible de déclarer une référence `j` sur une variable `i` : cela permet de créer un **nouveau nom `j` qui devient synonyme de `i`** (**un alias**).

```
int i = 10; // i est un entier valant 10
int &j = i; // j est une référence sur un entier, cet entier est i.
```

- La déclaration d'une référence se fait en précisant le type de l'objet référencé, puis le symbole `&`, et le nom de la variable référence qu'on crée.
- On pourra donc modifier le contenu de la variable en utilisant une référence (ici modifier `j` cela revient à modifier `i`).
- Une référence ne peut être initialisée qu'une seule fois : à la déclaration.
- Une référence ne peut donc référencer qu'une seule variable tout au long de sa durée de vie.
- Une référence ne peut pas être initialisée avec une valeur qu'une seule fois : `int &k = 44` est illégal.

En C++, on a la possibilité d'utiliser le **passage par référence** pour les paramètres. Cela est particulièrement intéressant lorsqu'on passe des objets en paramètre de fonctions ou méthodes et que le coût d'une recopie par valeur est trop important ("gros" objet) : on choisira alors un **passage par référence**.


```
class Entier
{
    private:
        int a;

    public:
        Entier(int a=0) : a(a) {}
        int getEntier() const { return a; }
        void setEntier(int i) { a = i; }
        void afficher() { cout << a << endl; }
};

Entier additionner(const Entier &e1, const Entier &e2)
{
    Entier entier;

    entier = e1.getEntier() + e2.getEntier();

    return entier;
}

void carre(Entier &e1)
{
    e1.setEntier(e1.getEntier() * e1.getEntier());
}

Entier e, entier1(2), entier2(5);

// passage des objets par référence constante
e = additionner(entier1, entier2); // soit 2 + 5
e.afficher(); // produit : 7

// passage d'objet par référence
carre(entier1); // soit 2 x 2
entier1.afficher(); // produit : 4
```

Dans l'exemple ci-dessus, les paramètres ne doivent pas être modifiés par la fonction `additionner()`, on a donc utilisé un passage par référence sur une variable constante (présence de `const`). C'est une bonne habitude à prendre ce qui permettra au compilateur de nous prévenir de toute tentative de modification :

```
Entier additionner(const Entier &e1, const Entier &e2)
{
    // Une belle erreur de programmation :
    e1.setEntier(e1.getEntier() + e2.getEntier()); // qui provoque heureusement une erreur
        de compilation
    return e1;
}
```


Destruction d'objets

Le **destructeur** est la **méthode membre appelée automatiquement lorsqu'une instance (objet) de classe cesse d'exister en mémoire** :

- Son rôle est de **libérer toutes les ressources qui ont été acquises lors de la construction** (typiquement libérer la mémoire qui a été allouée dynamiquement par cet objet).
- Un destructeur est une **méthode qui porte toujours le même nom que la classe, précédé de "~"**.
- Il existe quelques contraintes :
  - Il ne possède aucun paramètre.
  - Il n'y en a qu'un et un seul.
  - Il n'a jamais de type de retour.

La forme habituelle d'un destructeur est la suivante :

```
class T {
   public:
      ~T(); // destructeur
};
```

Pour éviter les fuites de mémoire, le destructeur d'une classe `PileChar` doit libérer la mémoire allouée au tableau de caractères `pile` :

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

   public:
      PileChar(int taille=50); // je suis le constructeur (par défaut) de la classe PileChar
      ~PileChar(); // je suis le destructeur de la classe PileChar
};

// Constructeur
PileChar::PileChar(int taille/*=50*/) : max(taille), sommet(0) // c'est la liste d'
    initialisation
{
   pile = new char[max]; // allocation dynamique du tableau de caractères
   cout << "PileChar(" << taille << ") : " << this << "\n";
}

// Destructeur
PileChar::~PileChar()
{
   delete [] pile; // libération mémoire du tableau de caractères
   cout << "~PileChar() : " << this << "\n";
}
```

PileChar.cpp

