La programmation OO :

« Cette partie est en cours de développement »

Sommaire

Sommaire

1. Principe de base et motivations de la POO

La vision du système=ensemble d'entité en interaction, objets, classifiés et regroupés en des classes : Exemple : nature, industrie, humain, …

2. Concept de bases :

2.1. Objet : état, comportement

2.2. Classe : attributs, méthodes

2.4. Syntaxe C++ et mécanisme DDU

Dans un seul fichier (non modulaire)

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

//déclartion

class point{
public:
      float x,y;
point();
};

//défintion

point::point(){
      x=0;y=0;
      cout<<"construction"<<endl;
}

//utilisation

int main() {
      cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
      point p;
      cout<<"objet créé"<<endl;
      //p.x=1;
      cout<<"p.x="<<p.x<<endl;
      return 0;
}
```

Mécanisme modulaire D.D.U en C++

En C++, un programme objet est divisé en trois parties:
Partie 1 : Déclaration : ici on déclare la classe, son nom, ses attributs et ses méthodes. Tout ça, se fera dans un fichier header portant le nom de « nom_classe.h »

Partie 2 : Définition : ici on définit l'implémentation de toutes les méthodes nécessaire. Tout ça, se fera dans un fichier dont le nom doit être « nom_classe.cpp » qui doit inclure l'autre fichier « nom_classe.h »
Partie 3 : Usage : qui est fait dans le programme principal main. Ce fichier doit inclure « nom_classe.h » aussi.

Exemple : de préférence de mettre name space=std lors de la création de la classe. Car c'est ce name space qui est utilisé automatiquement dans le projet main

---

```
//============================================================================
===
// Name        : POO2020_Mars.cpp
// Author      : p
// Version     :
// Copyright   : Your copyright notice
// Description : Hello World in C++, Ansi-style
//============================================================================
===
//utilisation

#include <iostream>
#include "Complexe.h"

using namespace std;

int main() {
	cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
	return 0;
}
```

```
//déclartion


/*
 * Complexe.h
 *
 *  Created on: 8 mars 2020
 *      Author: Sony
 */

#ifndef COMPLEXE_H_
#define COMPLEXE_H_

namespace std {

class Complexe {
public:
	Complexe();
	virtual ~Complexe();
};

} /* namespace std */

#endif /* COMPLEXE_H_ */
```

```
//défintion
```

```
/*
 * Complexe.cpp
 *
 *  Created on: 8 mars 2020
 *      Author: Sony
 */

#include "Complexe.h"

namespace std {

Complexe::Complexe() {
	// TODO Auto-generated constructor stub

}

Complexe::~Complexe() {
	// TODO Auto-generated destructor stub
}

} /* namespace std */
```

Exemples Simples :

- **Un point**
- **Un nombre complexe**

Les objets représentant des nombres complexes : Un nombre complexe est composé d'une partie réelle (re) et d'une partie imaginaire (im). Sur un nombre complexe on peut faire plusieurs opérations (méthodes). Lecture, écriture, …

2.3. Identification d'objets,

Exemples :

Identification d'objets :

Souvent depuis une description : les noms signifie les attributs et les verbes les méthodes

Exemple 1 :

"Ce marteau comporte un manche en bois, une extrémité plate en métal et une extrémité incurvée également en métal. Le manche permet de saisir le marteau, l'extrémité plate permet de frapper quelque chose et l'extrémité incurvée permet d'arracher quelque chose."

Exemple 2 :

"Ce chronomètre comporte un temps qui s'affiche et deux boutons A et B. Quand on presse sur A, on déclenche le chronomètre, ou bien on l'arrête. Quand on presse sur B, on remet à zéro le chronomètre."

3. Méthodes : constructeur, destructeur, consultateur (getters), modificateurs (setters), objet implicite (*this)

3.1. Constructeur :

Une méthode particulière permettant de créer des instances de la classe. Cette méthode peut faire l'initialisation des attributs. Il porte le même nom de la classe et il n'a pas de type de retour (même pas **void** comme le cas du C++). Si le programmeur ne définit pas un constructeur alors lors

de l'instanciation le constructeur par défaut (implicite et sans paramètre) est appelé pour créer une instance. Une fois le programmeur définit un constructeur, le constructeur implicite (ou par défaut) n'existera plus (et donc n'est plus accessible).

Exemples en C++

- exemple point, nombre complexe:
- Remarques sur les constructeur en C++

Remarque 1 : En C++, le constructeur assure une conversion de type :

Exemple :

```
Complexe::Complexe(float x) {
                re = x;
                im = 0.0;
                }
```

Dorénavant, toute déclaration :

```
Complexe z=1.0 ;
```

Génère un complexe dont la valeur de re est 1.0

D'où on convertie le **float** x en un complexe directement ;

Remarque 2 C++ : possibilité de redéfinir les opérateurs pour les objets de l'utilisateur

Exemple:

```
complex complex::operator +(cpx z){
        return cpx(x+z.x, y+z.y);
}
```

Remarque 3 C++ :

Possibilité d'initialiser les attributs avant les accolades des méthodes.

```
class C{
        //public:
        int x; int y;
        public:
        C(int x2);
        int get_X();
        int get_Y();
        ~C();
};
C::C(int x2):x(x2),y(0){
        //x=x2;
        //y=0;
}
```

3.2. Destructeur :

- Une méthode particulière qui permet de supprimer l'espace loué pour une instance déjà créée, après la fin de son usage.
-

Remarque: C++

- Le nom de constructeur est lui-même le nom de la classe précédé par le symbole ~ (Alt Gr 2). Exemple ~point ().
- On peut l'appeler explicitement ;
- On peut l'appeler même si il n'est pas ajouter explicitement ;

- On peut l'adapter pour faire un affichage ;

Exemple C++:

- En Java, l'utilisateur ne peut pas créer un destructeur. Le compilateur s'occupe de la tâche de suppression de l'espace mémoire, implicitement.

3.3. Setters & getters

4. Visibilité, portée, et encapsulation : pourquoi, exemple, …

La visibilité définit les zones du programme orienté objets où les attributs et les méthodes d'une classe sont reconnus. Il existe plusieurs visibilités possibles, par exemple: public, private, protected.

1) La visibilité **public** (public): Si un attribut (ou une méthode) d'une classe est public alors il est visible est accessible par tout dans le programme orienté objet.

Exemple en C++: Pour rendre un attribut ou une méthode public, il suffit de la précéder par le mot clé: `public:`

Dans le code suivant, tous le attributs et les méthodes de la classe C sont public.

```
class C{
      public:
      int x; int y;
      C(int x2);
      int get_X();
      int get_Y();
      ~C();
      };
```

Par conséquence, `x`, `y`, `get_X()`, `get_Y()`, et `~C()`, sont reconnus dans toutes les zones du code du programme: dans la définition des méthodes de la classe C, dans la définition des méthodes d'autre classes définies dans le même fichier où C est déclarée, et aussi dans le programme principal dans le fichier où C est déclarée. Si le programme est composé de plusieurs fichier et si la classe C est définit dans un fichier ".h", il suffit de faire inclure ce fichier ".h" pour avoir accès à tous les attributs et méthode de la classe C.

Exemple d'accès dans le main:

```
int main() {
      C obj1;
      obj1.x=1; // accès à x de obj1 possible car x est public
      obj1.y=2; // accès à y de obj1 possible car y est public
      cout<<obj1.get_X()<<endl;// accès à get_X() de obj1 possible car
                               //get_X() est public
      cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
      return 0;
}
```

2) La visibilité **private** (privé):

3) La visibilité **protected** (protégé):

exemple: C++

Notion d'encapsulation :

Séparer l'interface (services offerts par un objet) de l'implémentation (la réalisation) de cette interface. Pas d'accès externe

5. Relation entre classes : interaction (composition, agrégation), héritage (extension)

Exemple : Pour vérifier que nous n'avons rien oublié d'important dans une telle fiche descriptive, il faut imaginer l'objet à l'œuvre dans une petite scène. Le déroulement de l'action peut alors révéler des composants qui nous auraient échappé en première analyse (dans notre exemple, quatre acteurs: un marteau, un clou, un mur et un individu ; pour planter le clou dans le mur, l'individu saisit le marteau par son manche, puis frappe sur le clou avec l'extrémité plate ; il s'aperçoit alors que le clou est mal placé, et l'arrache avec l'extrémité incurvée).

5.1. Héritage : multiple (c++), simple (Java)

C++ :

Constructeur et destructeur :

1) Il **faut faire appel au constructeur de la classe mère si il existe pour initialiser les attributs de la classe mère (C++)** dans la définition du constructeur de la classe fille. (voir exemple en C++, ci-dessous)
2) L'appel du constructeur et du destructeur de la classe fille font appel implicitement et automatiquement aux constructeur/ destructeur de la classe mère. comme suit:
   Const_Mère, Const_fille, Dest_fille, Dest_Mère.
3) Composition
   Composante, composite, composite, composante

Méthodes virtuelle :

1) Une méthode définie (seulement) dans la classe mère, ensuite implémentée dans une classe fille.

Exemple 1 : héritage en C++: constructeur

```
class D:C{
      int z;
public:
      D(int x, int z2);
};
D::D(int x, int z2):C(x){// appel du constructeur de la classe mère
z2=0;
}
```

Exercice :

Déduire l'exécution du programme suivant :

```
#include <iostream>
using namespace std;
class C{
	//public:
	int x; int y; int id;
	public:
	C(int x2, int id0);
	int get_X();
	int get_Y();
	~C();
};
C::C(int x2, int id0):x(x2),y(0){

	//x=x2;
	//y=0;
	id=id0;
	cout<<"je suis créé, objet de la classe C id="<<id<<endl;
}
int C::get_X(){
	return x;
}
int C::get_Y(){
	return y;
}
C::~C(){
	cout<<"je suis mort, objet de la classe C id="<<id<<endl;
}

class D:C{
	int z;
public:
	D(int x, int z2, int id0);
};
D::D(int x, int z2, int id0):C(x, id0){
z2=0;
cout<<"je suis créé, objet de la classe D id="<<id0<<endl;
}

int main() {
	//C c1;
	//C c1=C(4);
	C c1=C(5, 1);
	//c1.x=3;
	//c1.C();
	c1.~C();
	D d=D(3,5, 2);
	cout<<c1.get_X()<<endl;
	cout<<c1.get_Y()<<endl;
	cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
	return 0;
}
```

Solution

```
je suis créé, objet de la classe C id=1
je suis mort, objet de la classe C id=1
je suis créé, objet de la classe C id=2
je suis créé, objet de la classe D id=2
5
0
!!!Hello World!!!
je suis mort, objet de la classe C id=2
je suis mort, objet de la classe C id=1
```

Remarque:

- la visibilité protected rend les attributs et méthodes définis dans la classe mère C reconnus dans les classes descendantes de C.
- Pour garder les mêmes visibilités des attributs et méthodes de C hérité par les objets de D, il faut précéder C par public lors de la création de D.
  exemple:

```
class C{
      //public:
      protected:
            int x; int y; int id;
      public:
            int get_X();
            int get_Y();
};

class D: public C{// pour que get_X(), get_Y() restent public pour D
      int z;
};
```

5.2. Composition & Agrégation : à développer dans les prochains jours

6. Concept Avancés :

6.1. Notion de Polymorphisme : surcharge, héritage, généricité,

1.1.1. Par surcharge :

Définir plusieurs méthodes ayant le même nom avec différentes signatures. Ceci dans la même classe, et donc accessibles toutes par les mêmes instances. Ce type existe même dans les langages impératifs (surcharge de fonctions).

Exemple : la classe Mere dispose de deux méthodes représentant une surcharge du nom M1()

```
class Mere{
public:
      //virtual
      void M1(){
            cout<<"M1()"<<endl;
      }
      void M1(int x){
            cout<<"M1(int x)"<<endl;
      }
};
```

Il est possible aussi de surcharger des opérateurs. Uniquement ces opérateur peuvent avoir 0 ou 1 argument.

Exemple : dans la classe complexe, on peut avoir deux implémentation de l'opérateur +.

```
complex complex::operator +(cpx z){
      return cpx(x+z.x, y+z.y);
}

complex complex::operator +(){
      return cpx(2+4, 5+6);
}
```

1.1.2. Par héritage :

Avant de commencer : Pour pouvoir accéder aux attributs et méthodes héritées par une classe fille de sa classe mère, il faut rendre cet héritage public (préserver les propriétés des attributs et méthodes hérités). On doit précéder le nom de la classe héritée par public lors de l'héritage.

```
class fille1: public Mere{
…
};
```

Deux cas sont possibles :

1) Sans méthode virtuelle

```
class Mere{
public:
   void M1(){
         cout<<"M1()"<<endl;
   }
};

class fille1: public Mere{
};

int main() {
    Mere m=Mere();
    m.M1();
    fille1 f0=fille1();
    cout<<"polymorphisme 1"<<endl;
    f0.M1();
    cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
      return 0;
   }
```

2) Avec méthode virtuelle

Considérant la situation suivante :

```
class Mere{
      void M();
};
void Mere::M(){
         cout<<"mère"<<endl;
      }

class fille1: public Mere{
public:
      void M(){
         cout<<"fille"<<endl;
      }
};
```

Si on a dans le programme principal :

```
int main() {
    Mere m=Mere();
    m.M();
    fille1 f0=fille1();
```

```
    f0.M1();
    Mere *f=new fille1();
    f->M();
    return 0;
}
```

Le résultat :

```
mère
fille
mère
```

Pourtant que f est construite par le constructeur de la classe fille, lors de l'appel elle fait appel à la méthode M() de la classe mère. Pour résoudre ce problème, il faut rendre M() de la classe mère une méthode virtuelle. Comme suit

```
class Mere{
public:

      virtual void M();
      //virtual ~Mere()=default; // de préférence ajouter un destructeur
virtuel
};
```

```
Maintenant, pour le même main précédent :
```

```
int main() {
    Mere m=Mere();
    m.M();
    fille1 f0=fille1();
    f0.M1();
    Mere *f=new fille1();
    f->M();
    return 0;
}
```

On aura l'affichage :

```
mère
fille
fille
```

Définition : une méthode virtuelle est une méthode définie dans sa classe mère, ensuite spécialisée dans les classes descendantes.

Définition : une méthode virtuelle pure est une méthode qui ne dispose pas de définition dans sa classe mère. Elle doit être définie dans les classes filles. Elle rend sa classe abstraite.

```
class Mere{
public:
      virtual void M()=0;
};
```

1.1.3. Par généricité :

6.2. Notion d'interface ou classe abstraite

- Une classe abstraite ne peut pas être instanciée: elle sert comme moule ou collection de noms d'attributs et méthodes qui peuvent être partagés entre plusieurs autre classes.
- Pour avoir des instances, on doit dériver une classe "fille" de cette classe ensuite faire des instanciations de la classe fille.

Exemple: C++:

- Une classe est dite abstraite, si elle contient au moins une méthode virtuelle pure
- Une méthode virtuelle est une méthode précédé par le mot clé: `virtual`
-

6.3. Notion de thread : objets actifs

- Communication synchrone entre objet.
- Communication asynchrone entre objet.