Classes C++

26 août 2024 12:01

Classe: définit les attributs et comportements communs à un ensemble d'objets.
Voir iFT232.

Instance: objet dont le type est de la classe, et a donc accès à ses attributs et fonctions.

```
class MaClasse
public:
        int x;      //mauvais

        void add(int val){
            x += val;
        }
};

MaClasse a;
MaClasse b;

a.x = 5;        // le . permet d'accéder aux
b.x = 10;       // attributs et fcts de l'objet

a.add(10);      //15

a.add(b.x);     //15+10=25
```

- 3 niveaux d'accès:
  - public: accessible partout

- 3 niveaux d'acces.
  public : accessible partout
  private : accessible seulement par le code dans MaClasse.
  (ou par les classe friend, voir notes)

  protected : par classes héritantes, voir IFT232

- Une fct de classe est souvent appelée une méthode.
- Bonne pratique:

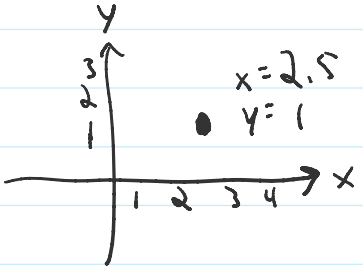
  public = méthodes correspondantes aux tâches de la structure de données

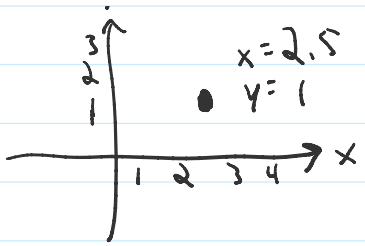
  private = tout ce qui est nécessaire pour implémenter ces tâches

Principe d'encapsulation : l'utilisateur d'une SDD ne doit pas avoir accès à l'implémentation, juste aux tâches à réaliser.

En principe, on pourrait changer l'implémentation et l'utilisateur ne verrait aucun changement.

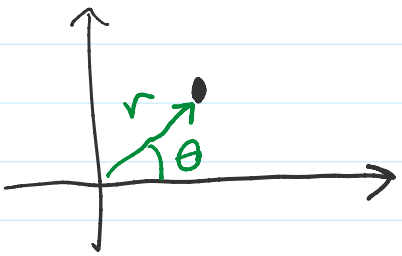
Exemple: Point dans l'espace 2D.






```
class Point{

private:
        double x,y;

public:
        void set_coord(double x2, double y2){

                x=x2;
                y=y2;
        }

        double get_x(){return x;}
        double get_y(){return y;}

};


Point p1;
p1.set_coord(1.5,2);
Point p2;
p2.set_coord(10,10);
cout<< p1.get_x();
```

On pourrait stocker le point en coordonnées polaires:



$x = r \cdot \cos\theta$
$y = r \cdot \sin\theta$

On change

```
private:
    double r, theta;

    get_x(){ return r* cos(theta); }
    get_y(){ return r* sin(theta); }
```

Constructeur: méthode spéciale appelée à la création de l'objet.

```
NomDeClasse(){
    //init
}

class Point{
...
public:
        Point(){
            x=0;
            y=0;
        }
};
Point p1; //appelle Point()
```

On peut définir plusieurs constructeurs,

On peut définir plusieurs constructeurs, avec arguments différents.

```
class Point{
    ...

        Point(){
            x=0;
            y=0;
        }

        Point(double x, double y){
            this->x=x;
            this->y=y;
        }
};

Point p1;
Point p2();
Point p3(10,10);

Point p1=Point();
Point p2=Point(10,10);
```

- Séparation des méthodes et implémentation

  Bonne pratique: la classe contient attribut et méthodes, leur implémentation vient ailleurs.

```
class Point{
```

```
class Point {
private:
        double x,y;
public:
        Point();
        Point(double x, double y);

        void set_coord(double x, double y);
        ...
};

// plus tard, au autre fichier

Point::Point(){
    x=0; y=0;
}

//voir .h
```

- Destructeur : méthode appelée quand l'objet est détruit.

```
~NomDeClasse(){  //code }


~Point(){
        // libérer la mémoire, au besoin
}
```

C++ détruit un objet lorsque sa portée est terminée.

portée est terminée.

Portée ≈ accolade { } contenant l'objet.

```
void f(){
    Point p1;
    {
        Point p2;
    } // p2 détruit
    {
        Point p1;
    } // p1 → détruit
} // p1 plus haut détruit
```

- Constructeur par copie

```
Point p1;
Point p2(p1);   // copie p1 dans p2
```

Si non-déclaré: copie les attributs de p1 dans p2.

Pour surchanger :

```
Point( Point& src){
    // copier src dans this
    x = src.x;
```

```
    // copier src dans this
    x = src.x;
    y = src.y;
}
```

- Assignation = (qui copie)

```
Point p1(10, 15);

Point p2;
p2 = p1;    // copie p1 dans p2
```

Par défaut : copie chaque attribut

Pour surcharger l'opérateur =

```
void operator=(Point& src){
    x = src.x;
    y = src.y;
}

p2 = p1; // appelle votre fonction =
```

- Le passage par valeur appelle Point(Point& src)

```
void f(Point p){...}

Point p1;
f(p1);    // p est créé via Point p(p1);
```

Bref, pour créer une classe C++, on détermine

- les attributs (souvent privés)
- les méthodes publiques, et privées
- les constructeurs
- le destructeur `~MaClasse`
- le copieur `MaClasse(MaClasse &src)`
- l'affectateur `operator=`

} souvent non nécessaire

Templates

`double x,y;` ← disons qu'on voudrait autre chose, par ex `int`.

Template: Pour utiliser un type que le user choisit.

```
template <typename TYPE>          // TYPE: nom du type,
class Point {                     // sera choisi par user
private:
        TYPE x;
        TYPE y;
public:
        Point();
        Point(TYPE x, TYPE y){
            this->x=x;
            this->y=y;
        }
```

```
};

Point<int> p1(10,10);
Point<double> p2(5.5, 2);
Point<string> pwtf("allo", "xyz");
```

Le compilateur va lire votre code et générer les classes

- Point_int
- Point_double
- Point_string