Object Oriented Programming - part 2

Emmanuelle Darles

Polymorphism in C++ language

Table of contents

Objectives

Goals of this lecture :

- Learn the concept of polymorphism in C++ language ;
- Design and implement robust and generic code.

I Class and object : Reminders

1. Class concept in oriented object programming

Definition

A class is a mold used to build objects. It is the grouping of characteristics and treatments within the same entity

It's an abstraction

Example

The class Point2D is used to build points in two dimensions

• Name: Point2D

• Features: x, y

• Treatments: display()

Definition: Attributes and methods

- The characteristics of a class are called attributes.
- The treatments are called methods.

Example:The class Point2D

```
#ifndef POINT2D_
#define POINT2D_

class Point2D
{
   public :

     // Constructors
     Point2D();
     Point2D(float x, float y);
     Point2D(const Point2D &P);

```

```cpp
    // Destructor
    ~Point2D();

    // Getters
    float getX() const;
    float getY() const;

    // Setter
    void setX(float x);
    void setY(float y);

    void display();

  private :

    // Attributes
    float x;
    float y;
};
#endif
```

```cpp
#include "point2D.hpp"
#include <iostream>

Point2D::Point2D()
{
   x = y = 0;
}
Point2D::Point2D(float x, float y)
{
   this->x = x;
   this->y = y;
}
Point2D::Point2D(const Point2D &P)
{
   this->x = P.x;
   this->y = P.y;
}
Point2D::~Point2D()
{
}
float Point2D::getX() const
{
   return this->x;
}
float Point2D::getY() const
{
   return this->y;
}
void Point2D::setX(float x)
{
   this->x = x;
}
void Point2D::setY(float y)
{
   this->y = y;
}
void Point2D::display()
```

```
{
    std::cout << "x = " << this->x << " y = " << this->y << std::endl;
}
```

2. Object concept in oriented object programming

Definition: Object

An object is an instance of a class.

Characteristics of an object

An object is defined by:

• a name: the name of the object

• one or more states (x=0, y=-1)

• one or more behaviors (treatments)

Fundamental:Static instanciation

Static instanciation consist in build an object by calling a class constructor implicitely.

This object is loaded in the stack.

```
#include "point2D.hpp"

int main()
{
    //Static instanciation
    Point2D P(-1,-1);
    P.display();
}
```

Fundamental:Dynamic instanciation

Dynamic instanciation consist in build an object by calling a constructor explicitely with the **`new`** operator.

This created object is a pointer loaded in the RAM.

To destruct the object, the destructor should be called explicitely with the **`delete`** operator.

```
#include "point2D.hpp"

int main()
{
    //Static instanciation
    Point2D P1(-1,-1);
    P1.display();

    //Dynamic instanciation
    Point2D *P2 = new Point2D(1,1);
    P2->display();
    delete(P2);
}
```

II Inheritance in C++

1. Inheritance concept

Definition: Inheritance

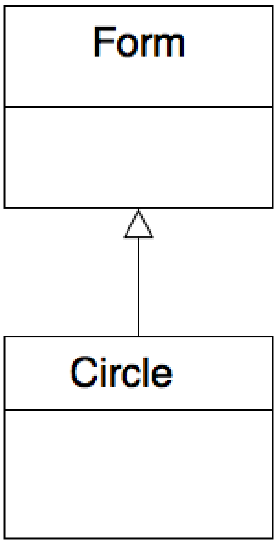
The inheritance allow to :

- code reuse
- add news fonctionnalities in an existing code
- modify an existing comportement by writting new definition of a method of a super class in a sub-class (*overriding*).

The inheritance allow to add some properties of an existing class to create a more precise class

The idea : a class B is a class A with more things

Example:Classes Form and Circle



```
#ifndef FORM_
#define FORM_

class Form
{
   public :

     // Constructors
     Form();

     // Destructor
     ~Form();

     void display();
};
#endif
```

```
#include "form.hpp"
#include <iostream>

Form::Form()
{
}
Form::~Form()
{
}
void Form::display()
{
   std::cout << "I'm a form !" << std::endl;
}
```

```
#ifndef CIRCLE_
#define CIRCLE_

#include "form.hpp"
#include "point2D.hpp"

class Circle : public Form
{
   public :
      Circle(Point2D center, float radius);
      ~Circle();

      //Getters
      Point2D getCenter();
      float   getRadius();

      //Setters
      void setCenter(Point2D center);
      void setRadius(float radius);

      //Overriding method
      void display();

    private :
        Point2D center;
        float radius;
};
#endif
```

```
#include "circle.hpp"
#include <iostream>

Circle::Circle(Point2D center, float radius) : Form()
{
    this->center = center;
    this->radius = radius;
}
Circle::~Circle()
{
}
Point2D Circle::getCenter()
{
    return this->center;
}
float Circle::getRadius()
{
    return this->radius;
}
void Circle::setCenter(Point2D center)
{
    this->center = center;
}
void Circle::setRadius(float radius)
{
    this->radius = radius;
}
void Circle::display()
{
   std::cout << "I'm a circle !" << std::endl;
   std::cout << "My center is ";
   this->center.display();
   std::cout << "My radius is " << this->radius << std::endl;
}
```

```
#include "point2D.hpp"
#include "circle.hpp"

int main()
{
    //Form creation
    Form f;
    f.display();

    //Circle creation
    Circle c(Point2D(0,0),1.0);
    c.display();
}
```

Output :

```
I'm a form !
I'm a circle !
My center is x = 0 y = 0
My radius is 1
```

An objet of type B can call all the method of the superclass A. If the method is override in the subclass B, it's possible to specify the method of the super-class with the : : operator

```
void Circle::display()
{
   Form::display();
}
```

Output :

```
I'm a form !
I'm a form !
```

But an object of type A can't call the methods of the subclass B.

```cpp
#include "point2D.hpp"
#include "circle.hpp"
#include <iostream>

int main()
{
    //Form creation
    Form f;
    f.display();

    //Circle creation
    Circle c(Point2D(0,0),1.0);
    c.display();

    std::cout << "radius : " << f.getRadius() << std::endl;
}
```

The last line produce an error during compilation ! A form is not a circle !

2. Constructor and destructor calling

Constructor calling

The super-class constructor is called in first.

Sub-class constructor is called secondly.

Example

```cpp
#include "point2D.hpp"
#include "circle.hpp"
#include <iostream>

int main()
{
    Circle c(Point2D(0,0),1.0);
    c.display();
}
```

- The constructor of the class Form is called.
- Then, the constructor of the class Circle is called.

Destructor calling

The destructor of the sub-class is called in first.

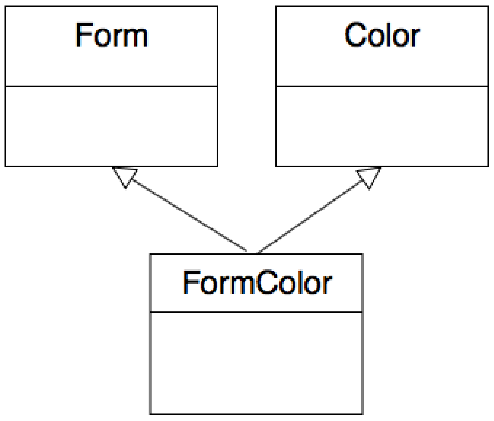
The destructor of the super-class is called secondly.

3. Multiple inheritance

Definition

In multiple inheritance, a sub-class has multiple super-classes. The sub-class inherits all the properties of the super-classes.

Example:Class Colored Form



```
#ifndef COLOR_
#define COLOR_

class Color
{
    public :

        //Constructors
        Color();
        Color(short red = 0, short green = 0, short blue = 0);

        //Destructor
        ~Color();

        //Getters
        short getRed() const;
        short getGreen() const;
        short getBlue() const;

        //Setters
        void setRed(short red);
        void setGreen(short green);
        void setBlue(short blue);

        void display();

    private :

        //Attributes
        short red, green, blue;
```

```cpp
};
#endif
```

```cpp
#include "color.hpp"
#include <iostream>

Color::Color(short r, short g, short b):red(r), green(g), blue(b)
{
}
Color::~Color()
{
}
short Color::getRed() const
{
    return this->red;
}
short Color::getGreen() const
{
    return this->green;
}
short Color::getBlue() const
{
    return this->blue;
}
void Color::display()
{
    std::cout << "I'm a color !" << std::endl;
    std::cout << "red: " << this->red << " green: " << this->green << " blue: " << this->blue
 << std::endl;
}
```

Example:Colored form

Consider a colored form class. This class inherits of two classes :

- the Color class ;
- and Form class.

```cpp
#ifndef FORM_COLOR_
#define FORM_COLOR_

#include "form.hpp"
#include "color.hpp"

class FormColor : public Form, public Color
{
    public :

        //Constructor
        FormColor(short r, short g, short b);

        //Destructor
        ~FormColor();

        void display();
};
#endif
```

```

```

```cpp
#include "formColor.hpp"

FormColor::FormColor(short r, short g, short b) : Form() , Color(r,g,b)
{
}
FormColor::~FormColor()
{
}
void FormColor::display()
{
    Form::display();
    Color::display();
}
```

```cpp
#include "formColor.hpp"

int main()
{
    //Colored form creation
    FormColor fc(10,10,10);
    fc.display();
}
```

Output :

```
I'm a form !
I'm a color !
red: 10 green: 10 blue: 10
```

4. Upcasting and Downcasting

Downcasting

The downcasting is to cast an object of type A (super-class) to an object of type B (sub-class)

```cpp
int main()
{
    Form *f = new Form();
    FormColor *fc = (FormColor*) (f);
    fc->display();
}
```

Output :

```
I'm a form !
I'm a color !
red: 0 green: 0 blue: 0
```

Upcasting

The upcasting is to cast an object of type B (sub-class) to an object of type A (super class)

```cpp
#include "formColor.hpp"

```

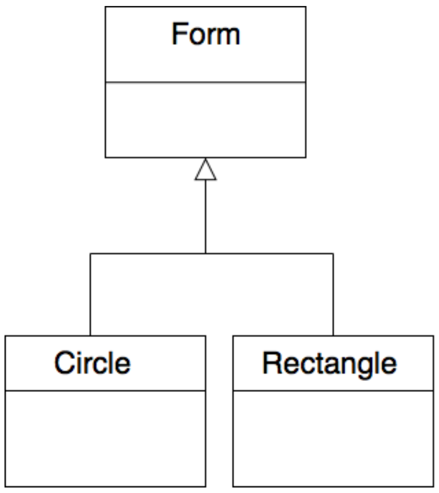
```cpp
int main()
{
    //Colored form creation
    FormColor fc(10,10,10);
    Form f = (Form) fc;
    f.display();
}
```

Output :

```
I'm a form !
```

Limitations

Consider the diagram below :



```cpp
#ifndef RECTANGLE_
#define RECTANGLE_

#include "form.hpp"
#include "point2D.hpp"

class Rectangle : public Form
{
   public :
      Rectangle(Point2D min, Point2D max);
      ~Rectangle();

      //Getters
      Point2D getMin() const;
      Point2D getMax() const;

      //Setters
      void setMin(Point2D min);
      void setMax(Point2D max);

      //Overriding method
      void display();

      //Overriding method
      float area();
```

```

    private :
        Point2D min;
        Point2D max;
};
#endif
```

```
#include "rectangle.hpp"
#include <math.h>
#include <iostream>

Rectangle::Rectangle(Point2D min, Point2D max)
{
    this->min = min;
    this->max = max;
}
Rectangle::~Rectangle()
{
}
Point2D Rectangle::getMin() const
{
    return this->min;
}
Point2D Rectangle::getMax() const
{
    return this->max;
}

void Rectangle::setMin(Point2D min)
{
    this->min = min;
}
void Rectangle::setMax(Point2D max)
{
    this->max = max;
}
void Rectangle::display()
{
   std::cout << "I'm a rectangle !" << std::endl;
   std::cout << "I'm defined by two points : ";
   this->min.display();
   this->max.display();
}
float Rectangle::area()
{
    return 2*fabs(this->min.getX()-this->max.getX())*fabs(this->min.getY()-this->max.getY());
}

```

Consider the code below :

```
#include "circle.hpp"
#include "rectangle.hpp"
#include <vector>

int main()
{
   std::vector<Form*> forms;
   Rectangle *R = new Rectangle(Point2D(-1,-1),Point2D(1,1));
   Circle *C = new Circle(Point2D(0,0),1);

   forms.push_back(R);
   forms.push_back(C);

   for(unsigned int i=0;i<forms.size();i++)
      forms[i]->display();
}
```

Output :

```
I'm a form !
I'm a form !
```

How to execute the good display method for each item of the collection ?

Solution : virtual methods !

III Polymorphism in C++

1. Polymorphism : what is it ?

At the top of a hierarchy, it is not always possible to give a general definition of some methods that are compatible with all subclasses.

For example :

- the method to display the form. It's not the same the method for a circle or a rectangle !
- the method to compute the area of a form. This method depend on the type of the form !

Definition: Polymorphism

Polymorphism is a way of manipulating heterogeneous objects in the same way, provided they have a common interface.

Polymorphism represents the ability of the system to dynamically choose the method that corresponds to the type of the object being manipulated.

A polymorphic object is an object that can take many forms during execution.

Polymorphism is implemented in C ++ with ***virtual functions and inheritance***.

2. Virtual function

Definition

A virtual function is a function that ***should be*** overridden in a subclass (different of ***that must be*** !).

It is possible to give in a subclass a definition of a general function defined in a super-class.

Example:Virtual method area()

```
#ifndef FORM_
#define FORM_

class Form
{
   public :

     // Constructors
     Form();

     // Destructor
     ~Form();

     virtual void display();

     virtual float area();

};
#endif
```

```
#include "form.hpp"
#include <iostream>

Form::Form()
{
}
Form::~Form()
{
}
void Form::display()
{
   std::cout << "I'm a form !" << std::endl;
}
float Form::display()
{
   //what we must return ?
   return 0;
}
```

```
#ifndef CIRCLE_
#define CIRCLE_

#include "form.hpp"
#include "point2D.hpp"

class Circle : public Form
{
   public :
      Circle(Point2D center, float radius);
      ~Circle();

      //Getters
      Point2D getCenter();
      float   getRadius();

      //Setters
      void setCenter(Point2D center);
      void setRadius(float radius);

      //Overriding method
      virtual void display() override;

      //Overriding method
      virtual float area() override;

```

```
    private :
        Point2D center;
        float radius;
};
#endif
```

```
#include "circle.hpp"
#include <iostream>
#include <math.h>

Circle::Circle(Point2D center, float radius) : Form()
{
    this->center = center;
    this->radius = radius;
}
Circle::~Circle()
{
}
Point2D Circle::getCenter()
{
    return this->center;
}
float Circle::getRadius()
{
    return this->radius;
}
void Circle::setCenter(Point2D center)
{
    this->center = center;
}
void Circle::setRadius(float radius)
{
    this->radius = radius;
}
void Circle::display() override
{
   std::cout << "I'm a circle !" << std::endl;
   std::cout << "My center is ";
   this->center.display();
   std::cout << "My radius is " << this->radius << std::endl;
}
float Circle::area() override
{
    return M_PI*this->radius*this->radius;
}
```

```
#ifndef RECTANGLE_
#define RECTANGLE_

#include "form.hpp"
#include "point2D.hpp"

class Rectangle : public Form
{
   public :
      Rectangle(Point2D min, Point2D max);
      ~Rectangle();

```

```
        //Getters
        Point2D getMin() const;
        Point2D getMax() const;

        //Setters
        void setMin(Point2D min);
        void setMax(Point2D max);

        //Overriding method
        void display() override;

        //Overriding method
        float area() override;

    private :
        Point2D min;
        Point2D max;
};
#endif
```

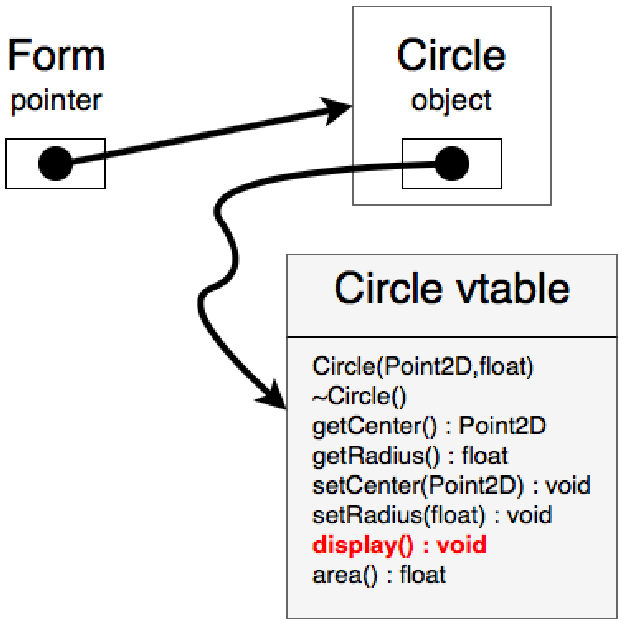
```
#include "rectangle.hpp"
#include <math.h>
#include <iostream>

Rectangle::Rectangle(Point2D min, Point2D max)
{
    this->min = min;
    this->max = max;
}
Rectangle::~Rectangle()
{
}
Point2D Rectangle::getMin() const
{
    return this->min;
}
Point2D Rectangle::getMax() const
{
    return this->max;
}

void Rectangle::setMin(Point2D min)
{
    this->min = min;
}
void Rectangle::setMax(Point2D max)
{
    this->max = max;
}
void Rectangle::display()
{
   std::cout << "I'm a rectangle !" << std::endl;
   std::cout << "I'm defined by two points : ";
   this->min.display();
   this->max.display();
}
float Rectangle::area()
{
    return 2*fabs(this->min.getX()-this->max.getX())*fabs(this->min.getY()-this->max.getY());
}

```

With virtual methods, the compilator can choose the good display method depending on the type of the object during the linking stage.



Example

```
#include "circle.hpp"
#include "rectangle.hpp"
#include <vector>

int main()
{
   std::vector<Form*> forms;
   Rectangle *R = new Rectangle(Point2D(-1,-1),Point2D(1,1));
   Circle *C = new Circle(Point2D(0,0),1);

   forms.push_back(R);
   forms.push_back(C);

   for(unsigned int i=0;i<forms.size();i++)
      forms[i]->display();
}
```

Output :

```
I'm a rectangle !
I'm defined by two points : x = -1 y = -1
x = 1 y = 1
I'm a circle !
My center is x = 0 y = 0
My radius is 1
```

3. Pure virtual method

Remember... At the top of a hierarchy, it is not always possible to give a general definition of some methods that are compatible with all subclasses.

Example

For example, the body of method area() in the class Form hasn't any sense ! Why return 0.0 ?

Definition

A pure virtual method makes it mandatory to define this method in the subclasses.

This method is declared in the superclass but it has no body in the superclass.

The body of this method (implementation) is defined in subclasses.

Example:The Form class with a display pure virtual method

```
#ifndef FORM_
#define FORM_

class Form
{
   public :

     // Constructors
     Form();

     // Destructor
     ~Form();

     virtual void display();

     virtual float area() = 0;

};
#endif
```

The method area() is a pure virtual method. She has no body in the super-class and must be overriden in subclasses.

Example:The Circle class

For subclasses, no changes ! Virtual methods are overriden.

Example:The main program :

```
#include "circle.hpp"
#include "rectangle.hpp"
#include <vector>
#include <iostream>
```

```cpp

int main()
{
   std::vector<Form*> forms;
   Rectangle *R = new Rectangle(Point2D(-1,-1),Point2D(1,1));
   Circle *C = new Circle(Point2D(0,0),1);

   forms.push_back(R);
   forms.push_back(C);

   for(unsigned int i=0;i<forms.size();i++){
       forms[i]->display();
       std::cout << "Area = " << forms[i]->area() << std::endl;
   }
}
```

Output :

```
I'm a rectangle !
I'm defined by two points : x = -1 y = -1
x = 1 y = 1
Area = 4
I'm a circle !
My center is x = 0 y = 0
My radius is 1
Area = 3.14159
```

4. Abstract class and Concrete class

Definition

An abstract class is a class wich contains one or more pure virtual methods.

Example

The class Form is an abstract class.

An abstract class cannot be instanciated because a pure virtual method has no body. During the linking stage, the method doesn't appear in the vtable.

Example

It's not possible to build form. It's possible to build circles, rectangles but not generic forms !

The code below produce an error :

```cpp
#include "form.hpp"

int main()
{
    Form F;
    F.display();
}
```

```
main.cpp:8:10: error: cannot declare variable 'F' to be of abstract type 'Form'
     Form F;
          ^
In file included from circle.hpp:4:0,
                 from main.cpp:1:
form.hpp:4:7: note:   because the following virtual functions are pure within 'Form':
 class Form
       ^
form.hpp:16:20: note:   virtual float Form::area()
      virtual float area() = 0;
```

Definition: Concrete class

A concrete class inherite of an abstract and override all pure virtual methods.

Example

The classes Circle and Rectangle are concrete classes. It's possible to build circles and rectangles.

```
#include "form.hpp"

int main()
{
    Circle C(Point2D(0,0),1);
    Rectangle R(Point2D(-1,-1),Point2D(1,1));
}
```

IV Quiz: The class Square

Download and decompress the archive **`forms.zip`** in UpDago.

Test all the classes by using the main program (main.cpp)

[cf. forms]

To compile in command line (in a terminal) :

```
g++ -std=c++11 circle.cpp rectangle.cpp form.cpp point2D.cpp main.cpp -o forms
```

To execute (in a terminal) :

```
./forms
```

Question 1

We consider square objects defined by one point in 2D (the center) and a float representing the length of the sides.

A square is display as below :

```
I'm a square !
My center is :
x = 0 y = 0
The length of my sides is 2.0
```

Implement the class Square. You must override all the pure virtual methods of the class Form.

The area of a square can be calculated by the square of the length of the sides.

Test your class Square with the program below :

```
#include "circle.hpp"
#include "rectangle.hpp"
#include "square.hpp"
#include <vector>
#include <iostream>

int main()
{
   std::vector<Form*> forms;
   Rectangle *R = new Rectangle(Point2D(-1,-1),Point2D(1,1));
   Circle *C = new Circle(Point2D(0,0),1);
   Square *S = new Square(Point2D(0,0),2);

   forms.push_back(R);
   forms.push_back(C);
   forms.push_back(S);

   for(unsigned int i=0;i<forms.size();i++){
       forms[i]->display();
       std::cout << "Area = " << forms[i]->area() << std::endl;
   }
}
```

Question 2

We consider triangle objects defined by three points in 2D.

A triangle is display as below :

```
I'm a triangle
My points are :
x = 0 y = 0
x = 0 y = 1
x = 1 y = 0
```

Implement the class Triangle. You must override all the pure virtual methods of the class Form.

V Quiz: Crazy robots

Question 1

Implement an abstract class Robot with a pure virtual method go().

A robot is defined by a position (a point in 2D) and a list of positions covered by the robot.

Question 2

Implement a class Crazy Robot which inherit of the class Robot.

A crazy robot correspond to a robot which move in an aleatory way. For example, it can go to the left one time and go up and then right. Its displacement is purely aleatory (he goes in all directions and he's just crazy !)