Short Course on Programming in C/C++

Organized by Onur Pekcan

Contributor Selim Temizer Instructor Hasan Yılmaz

Week 2 – Lecture2

Today

We will cover;

- **Introduction to C++ and Object Oriented Programming(cont.)**
  - Friendship and Inheritance
  - Polymorphism

Friendship

Friend Functions

- In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not affect *friends*.

- Friends are functions or classes declared with the friend keyword.

- If we want to declare an external function as friend of a class, thus allowing this function to have access to the private and protected members of this class, we do it by declaring a prototype of this external function within the class, and preceding it with the keyword friend:

Example

```
// friend functions
#include <iostream>
using namespace std;

class CRectangle {
   int width, height;

   public:
   void set_values (int, int);
   int area () {return (width * height);}
   friend CRectangle duplicate (CRectangle);
};

void CRectangle::set_values (int a, int b) {
  width = a;
  height = b;
}
```

Example(cont.)

```
CRectangle duplicate (CRectangle rectparam)
{
  CRectangle rectres;
  rectres.width = rectparam.width*2;
  rectres.height = rectparam.height*2;
  return (rectres);
}

int main () {
  CRectangle rect, rectb;
  rect.set_values (2,3);
  rectb = duplicate (rect);
  cout << rectb.area();
  return 0;
}
```

Output:
24

- duplicate function simply has access to its private and protected members without being a member.

Friendship

Friend Classes

- We can also define a class as friend of another one, granting that first class access to the protected and private members of the second one.

Example

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

class CSquare;

class CRectangle {
    int width, height;
  public:
    int area ()
      {return (width * height);}
    void convert (CSquare a);
};

class CSquare {
  private:
    int side;
  public:
    void set_side (int a)
      {side=a;}
    friend class CRectangle;
};
```

```
void CRectangle::convert (CSquare a) {
  width = a.side;
  height = a.side;
}

int main () {
  CSquare sqr;
  CRectangle rect;
  sqr.set_side(4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

Output:

16

Inheritance

- **Inheritance** is the fundamental object-oriented principle governing the reuse of code among related classes.
- Inheritance models the **IS-A relationship**. In an IS-A relationship, the derived class is a variation of the base class.
  - e.g. Circle IS-A Shape, car IS-A vehicle.
- Using inheritance a programmer creates new classes from an existing class by adding additional data or new functions, or by redefining functions.

Inheritance Hierarchy

- Inheritance allows the derivation of classes from a **base class** without disturbing the implementation of the base class.
- A **derived class** is a completely new class that inherits the properties, public methods, and implementations of the base class.
- The use of inheritance typically generates a **hierarchy** of classes.
- In this hierarchy, the derived class is a **subclass** of the base class and the base class is a **superclass** of the derived class.
- These relationships are **transitive**.

Base and Derived Classes

- Often an object from a derived class (subclass) “is an” object of a base class (superclass)

Base class	Derived classes
`Student`	`GraduateStudent` `UndergraduateStudent`

`Shape`	`Circle`
	`Triangle`
	`Rectangle`

`Loan`	`CarLoan`
	`HomeImprovementLoan`
	`MortgageLoan`

`Employee`	`FacultyMember` `StaffMember`
`Account`	`CheckingAccount` `SavingsAccount`

Syntax

class derived_class_name: **public** base_class_name
{
        /*...*/
};

class Crectangle: ***public*** CPolygon { ... }

- This **public** keyword after the colon (:) denotes the most accessible level the members inherited from the class that follows it (in this case CPolygon) will have. Since public is the most accessible level, by specifying this keyword the derived class will inherit all the members with the same levels they had in the base class.

Example

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

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };

class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

Output:

20

10

Another Example of Inheritance

```
class mammal   // base class
{
   public:
     // manager functions
     mammal( int age = 0, int wt = 0 ):itsAge(age),
              itsWt( wt ) { }
     ~mammal() { }

     // access functions
     int getAge() const { return itsAge; }
     int getWt() const { return itsWt; }

     // implementation functions
     void speak() const{ cout << "mammal sound!\n";}
     void sleep() const{ cout << zzzzzzzzzzz!\n"; }

   protected:
     int itsAge, itsWt;
};
```

```
class dog : public mammal
{
   public:
     // manager functions
     dog( int age, int wt, string name ) :
     mammal( age, wt )
                  { itsName = name; }
     dog( int age=0, int wt=0 ) : mammal(age,wt)
                  { itsName = ""; }
     ~dog() { } // nothing to do

     // implementation function
     void speak() const { cout << "ARF ARF\n"; }
     void wagtail() const { cout << "wag wag wag\n"; }

   private:
     string itsName;
};
```

```
int main()
{
   dog bowser(3, 25, "Bowser");

   bowser.speak();
   bowser.mammal :: speak();
   bowser.wagtail();
   bowser.sleep();

   cout << "bowser is " << bowser.getAge() << " years old!" <<
endl;
   return 0;
}
```

Here is the output of the sample code:

```
ARF ARF
mammal sound!
wag wag wag
zzzzzzzzzzz!
bowser is 3 years old!
```

Overriding Functions

- If derived class has a member function with the same name, return type and parameter list as in the base class, then the derived class function *overrides* the base class function.
- The base class function is *hidden*.
- The implementation of the base class function has been changed by the derived class.
- Derived class objects invoke the derived version of the function.
- If a derived class object wants to use the base class version, then it can do so by using the scope resolution operator:

```
bowser.speak()   // derived class version is invoked
bowser.mammal::speak() //base class version
```

Private vs protected class members

1. private base class member(s)
   - derived class member functions can not access these objects directly
   - the member still exists in the derived class object
   - because not directly accessible in the derived class, the derived class object must use base class access functions to access them

2. protected base class member(s)
   - directly accessible in the derived class
   - member becomes a protected member of the derived class as well

Constructors and destructors

1. Constructors
   - Constructors are not inherited.
   - Base class constructor is called before the derived class constructor (either explicitly, or if not then the compiler invokes the default constructor).
   - Base class constructor initializes the base class members.
   - The derived class constructor initializes the derived class members that are not in the base class.
   - A derived class constructor can pass parameters to the base class constructor as illustrated in the example.
   - Rules of thumb for constructors under inheritance:
     - Define a default constructor for every class.
     - Derived class constructors should explicitly invoke one of the base class constructors.
2. Destructors
   - Derived class destructor is called before the base class destructor.
   - Derived class destructor does cleanup chores for the derived class members that are not in the base class.
   - Base class destructor does the same chores for the base class members.

Abstract Methods and Classes

- Delete this topic
- An **abstract method** is declared in the base class and always defined in the derived class.
- It does not provide a default implementation, so each derived class must provide its own implementation.
- A class that has at least one abstract method is called an **abstract class**.
- Abstract classes can never be instantiated.

Example

- An abstract class : `Shape`
- Derive specific shapes: `Circle, Rectangle`
- Derive `Square` from `Rectangle`
- The `Shape` class can have data members that are common to all classes:e.g. `name, positionOf.`
- Abstract methods apply for each particular type of object: e.g. `area`

Abstract base class Shape

```
class shape
{
   public:
       Shape(const string & shapeName = “”)
          : name( shapeName ) {}
       virtual ~Shape( ) { }
       virtual double area( ) const = 0;
       bool operator< (const Shape & rhs) const
          { return area () < rhs.area (); }
       virtual void print(ostream & out = cout ) const
          { out << name << “ of area “ << area();}
   private:
       string name;
}
```

Expanding Shape Class

```
const double PI = 3.1415927;

class Circle : public Shape
{
   public :
        Circle( double rad = 0.0 )
           : Shape("circle"), radius(rad) {}
        double area() const
           { return PI * radius * radius;}
   private:
        double radius;
};
```

Accessing the Variables of a Class

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived classes	yes	yes	no
not members	yes	no	no

Polymorphism

- Before getting into this section, it is recommended that you have a proper understanding of pointers and class inheritance. If any of the following statements seem strange to you, you should review the indicated sections:

Statement:	**Explained in:**
int a::b(int c) { }	Classes
a->b	Data Structures
class a: public b { };	Friendship and inheritance

Pointers to Base Class

- One of the key features of derived classes is that a pointer to a derived class is type-compatible with a pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature, that brings Object Oriented Methodologies to its full potential.

Example

```
// pointers to base class
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
  };

class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

Output:

```
20
10
```

Pointers to Base Class

- In function main, we create two pointers that point to objects of class CPolygon (ppoly1 and ppoly2). Then we assign references to rect and trgl to these pointers, and because both are objects of classes derived from CPolygon, both are valid assignment operations

- The only limitation in using *ppoly1 and *ppoly2 instead of rect and trgl is that both *ppoly1 and *ppoly2 are of type CPolygon* and therefore we can only use these pointers to refer to the members that CRectangle and CTriangle inherit from CPolygon. For that reason when we call the area() members at the end of the program we have had to use directly the objects rect and trgl instead of the pointers *ppoly1 and *ppoly2

Pointers to Base Class

- In order to use area() with the pointers to class CPolygon, this member should also have been declared in the class CPolygon, and not only in its derived classes, but the problem is that CRectangle and CTriangle implement different versions of area, therefore we cannot implement it in the base class

Virtual Members

- A member of a class that can be redefined in its derived classes is known as a virtual member. In order to declare a member of a class as virtual, we must precede its declaration with the keyword virtual:

Example

```
// virtual members
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area ()
      { return (0); }
  };

class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };
```

```
int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon poly;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  CPolygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  cout << ppoly3->area() << endl;
  return 0;
}
```

Output:

20

10

0

Virtual Members

- The member function area() has been declared as virtual in the base class because it is later redefined in each derived class. You can verify if you want that if you remove this virtual keyword from the declaration of area() within CPolygon, and then you run the program the result will be 0 for the three polygons instead of 20, 10 and 0. That is because instead of calling the corresponding area() function for each object (CRectangle::area(), CTriangle::area() and CPolygon::area(), respectively), CPolygon::area() will be called in all cases since the calls are via a pointer whose type is CPolygon*.

- A class that declares or inherits a virtual function is called a ***polymorphic class***

Abstact Base Classes

- Abstract base classes are something very similar to our CPolygon class of our previous example. The only difference is that in our previous example we have defined a valid area() function with a minimal functionality for objects that were of class CPolygon (like the object poly), whereas in an abstract base classes we could leave that area() member function without implementation at all.

- This is done by appending =0 (equal to zero) to the function declaration.

Example

```
// abstract class CPolygon
class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area () = 0;
};
```

- This type of function is called a ***pure virtual function***, and all classes that contain at least one pure virtual function are ***abstract base classes***.

Abstact Base Classes

- The main difference between an abstract base class and a regular polymorphic class is that because in abstract base classes at least one of its members lacks implementation we **cannot** create instances (objects) of it
- But a class that cannot instantiate objects is not totally useless. We can create pointers to it and take advantage of all its polymorphic abilities. Therefore a declaration like:

CPolygon poly;

would not be valid for the abstract base class we have just declared, because tries to instantiate an object. Nevertheless, the following pointers:

CPolygon * ppoly1;
CPolygon * ppoly2;

would be perfectly valid

Example

```
// abstract base class
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
  };

class CRectangle: public CPolygon {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area (void)
      { return (width * height / 2); }
  };
int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  return 0;
}
```

Output:

20

10