Lecture-16

Classes (I)

16.1 Definition :A *class* is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.

An *object* is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are generally declared using the keyword `class`, with the following format:

```
class class_name {
  access_specifier_1://private, public and protected
    member1;
  access_specifier_2:
    member2;
  ...
  } object_names;
```

where class_name is a valid identifier for the class, object_names is an optional list of names for objects of this class. The body of the declaration can contain members, that can be either data or function declarations, and optionally access specifiers.

All is very similar to the declaration on data structures, except that we can now include also functions and members, but also this new thing called *access specifier*. An access specifier is one of the following three keywords: private, public or protected. These specifiers modify the access rights that the members following them acquire:

- private members of a class are accessible only from within other members of the same class or from their *friends*.
- protected members are accessible from members of their same class and from their friends, but also from members of their derived classes.
- Finally, public members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the class keyword have private access for all its members. Therefore, any member that is declared before one other class specifier automatically has private access. For example:

```
class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
```

```
    int area (void);
  } rect;
```

declares a class (i.e., a type) called CRectangle and an object (i.e., a variable) of this class called rect. This class contains four members: two data members of type int (member x and member y) with private access (because private is the default access level) and two member functions with public access: set_values() and area(), of which for now we have only included their declaration, not their definition.

Notice the difference between the class name and the object name: In the previous example, CRectangle was the class name (i.e., the type), whereas rect was an object of type CRectangle. It is the same relationship int and a have in the following declaration:

int a;

where int is the type name (the class) and a is the variable name (the object).

After the previous declarations of CRectangle and rect, we can refer within the body of the program to any of the public members of the object rect as if they were normal functions or normal variables, just by putting the object's name followed by a dot (.) and then the name of the member. All very similar to what we did with plain data structures before. For example:

```
rect.set_values (3,4);
myarea = rect.area();
```

The only members of rect that we cannot access from the body of our program outside the class are x and y, since they have private access and they can only be referred from within other members of that same class.

Here is the complete example of class CRectangle:

Program 16.1: Scope resolutionOperator

```
// classes example
#include <iostream>
using namespace std;

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area () {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
```

```
area: 12
```

```
  x = a;
  y = b;
}

int main () {
  CRectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

The most important new thing in this code is the operator of scope (::, two colons), also called **scope resolution operator**, included in the definition of set_values(). It is used to define a member of a class from outside the class declaration itself.

You may notice that the definition of the member function area() has been included directly within the definition of the CRectangle class given its extreme simplicity, whereas set_values() has only its prototype declared within the class, but its definition is outside it. In this outside declaration, we must use the operator of scope (::) to specify that we are defining a function that is a member of the class CRectangle and not a regular global function.

The scope operator (::) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. For example, in the function set_values() of the previous code, we have been able to use the variables x and y, which are private members of class CRectangle, which means they are only accessible from other members of their class.

The only difference between defining a class member function completely within its class and to include only the prototype and later its definition, is that in the first case the function will automatically be considered an inline member function by the compiler, while in the second it will be a normal (not-inline) class member function, which in fact supposes no difference in behavior.

Members x and y have private access (remember that if nothing else is said, all members of a class defined with keyword class have private access). By declaring them private we deny access to them from anywhere outside the class. This makes sense, since we have already defined a member function to set values for those members within the object: the member function set_values(). Therefore, the rest of the program does not need to have direct access to them. Perhaps in a so simple example as this, it is difficult to see an utility in protecting those two variables, but in greater projects it may be very important that values cannot be modified in an unexpected way (unexpected from the point of view of the object).

One of the greater advantages of a class is that, as any other type, we can declare several objects of it. For example, following with the previous example of class CRectangle, we could have declared the object rectb in addition to the object rect:

Program 16.2: Example: one class, two objects

```
// example: one class, two objects
#include <iostream>
using namespace std;

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area () {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}

int main () {
  CRectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

```
rect area: 12
rectb area: 30
```

In this concrete case, the class (type of the objects) to which we are talking about is CRectangle, of which there are two instances or objects: rect and rectb. Each one of them has its own member variables and member functions.

Notice that the call to rect.area() does not give the same result as the call to rectb.area(). This is because each object of class CRectangle has its own variables x and y, as they, in some way, have also their own function members set_value() and area() that each uses its object's own variables to operate.

That is the basic concept of *object-oriented programming*: Data and functions are both members of the object. We no longer use sets of global variables that we pass from one function to another as parameters, but instead we handle objects that have their own data and functions embedded as members. Notice that we have not had to give any parameters in any of the calls to rect.area or rectb.area. Those member functions directly used the data members of their respective objects rect and rectb.

16.2 Constructors and destructors

Objects generally need to initialize variables or assign dynamic memory during their process of creation to become operative and to avoid returning unexpected values during their execution. For example, what would happen if in the previous example we called the member function area() before having called function set_values()? Probably we would have gotten an undetermined result since the members x and y would have never been assigned a value.

In order to avoid that, a class can include a special function called constructor, which is automatically called whenever a new object of this class is created. This constructor function must have the same name as the class, and cannot have any return type; not even void.

We are going to implement CRectangle including a constructor:

Program 16.3 Constructors

```
// example: class constructor
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
  public:          //public function can access  private numbers
    CRectangle (int,int);
    int area () {return (width*height);}
};

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

int main () {
  CRectangle rect (3,4);//Object can access only public function
  CRectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

```
rect
area:
12
rectb
area:
30
```

As you can see, the result of this example is identical to the previous one. But now we have removed the member function set_values(), and have included instead a constructor that performs a similar action: it initializes the values of x and y with the parameters that are passed to it.

Notice how these arguments are passed to the constructor at the moment at which the objects of this class are created:

```
CRectangle rect (3,4);
CRectangle rectb (5,6);
```

Constructors cannot be called explicitly as if they were regular member functions. They are only executed when a new object of that class is created.

You can also see how neither the constructor prototype declaration (within the class) nor the latter constructor definition include a return value; not even void.

The *destructor* fulfills the opposite functionality. It is automatically called when an object is destroyed, either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using the operator delete.

The destructor must have the same name as the class, but preceded with a tilde sign (~) and it must also return no value.

The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated.

Program 16.4 ; Constructors and Destructors

```
// example on constructors and destructors
#include <iostream>
using namespace std;

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

CRectangle::CRectangle (int a, int b) {
  width = new int;
 /* Operator new is to allocate new block of memory to a
pointer in dynamic conditions */
  height = new int;
  *width = a;
  *height = b;
}

CRectangle::~CRectangle () {
  delete width;
  delete height;
}
```

```
rect area:
12
rectb
area: 30
```

```
int main () {
  CRectangle rect (3,4), rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

The constructor functions have some special characteristics. These are

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefore can not return any value.
- They can not be **inherited**, though the derived class can call the base class constructor.
- Like other C++ functions, they can have **default arguments**.
- Constructors can not be **virtual.**
- We can not refer to their addresses.
- An object with a constructor or destructor can not be used as a member of a **union.**
- They make 'implicit calls' to the operator **new** and **delete** when memory allocation is reqired.

16. 3 Parameterized constructors.

The constructors *that can take arguments* are called parameterized constructors.

The constructor **integer()** below may be modified to take arguments as shown below.

```
Class integer
     {
     int m,n;
     public:
       integer(int x, int y);
       ……….
      ……..…..
};
 integer :: integer(int x, int y)
{
   m= x;
    n =y;
}
```

When the constructor has been parameterized, the object declaration statement such as

Integer int1;

may not work. Initial values are passed as arguments to the constructor function, when an object is declared in two ways:

- By calling the constructor explicitly.
- By calling the constructor implicitly.

The following declaration illustrates the two methods:

integer int1 = integer(0,100); //explicit call

integer int1(0,100); //implicit call

Implicit call to constructor. By implicit call, it means that the constructor is called (invoked) even when its name has not been mentioned in the statement. When name is mentioned in call, it is called explicit.

16.4 Temporary Instances. The *explicit call* to a constructor also allows you to create a *temporary instance* or *temporary object.* A temporary instance is the one that lives in the memory as long as it is being used or referenced in an expression and after this it dies. The temporary instances are anonymous i.e. they do not bear a name.

Following code illustrates the temporary instance.

```
class sample { int i,j;
public:
   sample (int a, int b)
{   i=a; j=b; }
void print (void)
{ cout<<i<<j<<" \n";
}
……
……..
};
void main (void)
{
sample S1(2,5);   //An object S1 is created
          S1.print();    // data value of S1 is printed
          sample(4,9).print();    //Data values of a temporary
                                  //sample instance printed
 }
```

16.5 Overloading Constructors

Like any other function, a constructor can also be overloaded with more than one function that have the same name but different types or number of parameters. Remember that for overloaded functions the compiler will call the one whose parameters match the arguments used in the function call. In the case of constructors, which are automatically called when an object is created, the one executed is the one that matches the arguments passed on the object declaration:

Program 16.5 : Overloading of constructors

```
// overloading class constructors
#include <iostream>
using namespace std;

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

CRectangle::CRectangle () {
  width = 5;
  height = 5;
}

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

int main () {
  CRectangle rect (3,4);
  CRectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

```
rect area: 12
rectb area: 25
```

In this case, rectb was declared without any arguments, so it has been initialized with the constructor that has no parameters, which initializes both width and height with a value of 5.

Important: Notice how if we declare a new object and we want to use its default constructor (the one without parameters), we do not include parentheses ():

```
CRectangle rectb;   // right
CRectangle rectb(); // wrong!
```

16.6 Default constructor

If you do not declare any constructors in a class definition, the compiler assumes the class to have a *default constructor* with no arguments. Therefore, after declaring a class like this one:

```
class CExample {
  public:
    int a,b,c;
    void multiply (int n, int m) { a=n; b=m; c=a*b; };
  };
```

The compiler assumes that `CExample` has a default constructor, so you can declare objects of this class by simply declaring them without any arguments:

```
CExample ex;
```

But as soon as you declare your own constructor for a class, the compiler no longer provides an implicit default constructor. So you have to declare all objects of that class according to the constructor prototypes you defined for the class:

```
class CExample {
  public:
    int a,b,c;
    CExample (int n, int m) { a=n; b=m; };
    void multiply () { c=a*b; };
  };
```

Here we have declared a constructor that takes two parameters of type int. Therefore the following object declaration would be correct:

```
CExample ex (2,3);
```

But,

```
CExample ex;
```

would not be correct, since we have declared the class to have an explicit constructor, thus replacing the default constructor.

But the compiler not only creates a default constructor for you if you do not specify your own. It provides *three special member functions* in total that are implicitly declared if you do not declare your own. These are the *copy constructor*, the *copy assignment operator*, and the default destructor.

16.7 The copy constructor and the *copy assignment operator* copy all the data contained in another object to the data members of the current object. For CExample, the copy constructor implicitly declared by the compiler would be something similar to:

```
CExample::CExample (const CExample& rv) {
  a=rv.a;  b=rv.b;  c=rv.c;
  }
```

Therefore, the two following object declarations would be correct:

```
CExample ex (2,3);
CExample ex2 (ex);   // copy constructor (data copied from ex)
CExample ex3=ex; //ex copies to ex3. Copy constructor called again
```

Program 16.6 Example of Copy constructor

```
//Example of Copy constructor

#include <iostream>
using namespace std;

class code
{
      int id;
public:
      code(){ }     //constructor
      code(int a){id=a;}//constructor again
      code (  code & x )   //copy constructor
{

        id=x.id;// copy in the value
}

void display(void)
{
      cout<<id;
}
};
int main()
{
      code A(100); //Object A is created and initialised
      code B(A); //copy constructor called
      code C=A;  //copy constructor called again

      code D; //D is created not initialised
            D=A;  //copy constructor not called

      cout<<"\n id of A: "; A.display();
      cout<<"\n id of B: "; B.display();
      cout<<"\n id of C: " ;C.display();
      cout<<"\n id of D: " ;D.display();

      return 0;
}
/*
id of A: 100
 id of B: 100
 id of C: 100
 id of D: 100
*/
```

Copy constructor is

- a constructor function with the same name as the class
- used to make deep copy of objects.

There are 3 important places where a copy constructor is called.

1. When an object is created from another object of the same type
2. When an object is passed *by value* as a parameter to a function
3. When an object is returned from a function

If a copy constructor is not defined in a class, the compiler itself defines one. This will ensure a shallow copy. If the class does not have pointer variables with dynamically allocated memory, then one need not worry about defining a œopy constructor. It can be left to the compiler's discretion.

But if the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor.

For ex:

```
class A   //Without copy constructor
{
    private:
    int x;
    public:
    A() {A = 10;}
    ~A() {}
}
```

```
class B    //With copy constructor
{
    private:
    char *name;
    public:
    B()
    {
    name = new char[20];
    }
    ~B()
    {
    delete name[];
    }
//Copy constructor
    B(const B &b)
    {
```

```
        name = new char[20];
        strcpy(name, b.name);
        }
    };
```

Let us Imagine if you don't have a copy constructor for the class B. At the first place, if an object is created from some existing object, we cannot be sure that the memory is allocated. Also, if the memory is deleted in destructor, the delete operator might be called twice for the same memory location.

This is a major risk. One happy thing is, if the class is not so complex this will come to the fore during development itself. But if the class is very complicated, then these kind of errors will be difficult to track.

In Windows this will lead to an application popup and unix will issue a core dump. A careful handling of this will avoid a lot of nuisance.

16.8 The constructor functions can also be defined as inline **functions. Example**

```
class integer {
Int m,n;
public:
integer (int x, int y)  //inline constructors
{
m=x;
n=y;
}
…..
…..
};
```

The parameters of a constructor can be of any type except that of the class to which it belongs. For example,

```
class integer {
int m,n;
public:
integer (integer)
};
```

However, a constructor can accept a *reference* to its own class as a parameter. Thus the statement,

```
class integer {
int m,n;
public:
```

integer (integer&)

};

is valid. In such a case constructor is called ***copy constructor.***

Following program demonstrate the overloading of constructors**.**

16.9 Friend functions

In principle, private and protected members (protected member is accessible by the member functions within its class and any class immediately derived from it.) 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 as such.

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:

Program 16.9: Friend function

```
// 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;
}

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;
}
```

The duplicate function is a friend of CRectangle. From within that function we have been able to access the members width and height of different objects of type CRectangle, which are private members. Notice that neither in the declaration of duplicate() nor in its later use in main() have we considered duplicate a member of class CRectangle. It isn't! It simply has access to its private and protected members without being a member.

The friend functions can serve, for example, to conduct operations between two different classes. Generally, the use of friend functions is out of an object-oriented programming methodology, so whenever possible it is better to use members of the same class to perform operations with them. Such as in the previous example, it would have been shorter to integrate duplicate() within the class CRectangle