http://uranchimeg.com

Object oriented programming C++

T.Uranchimeg
Prof. Dr.

Email
uranchimeg@must.edu.mn

Power Engineering School

Subjects

- *Class and Object*
- *Declare a class*
- *Constructors*
- *Destructors*

A class

A class is a logical method to organize data and functions in the same structure. They are declared using keyword **class**, whose functionality is similar to that of the C keyword **struct**, but with the possibility of including functions as members, instead of only data.

Its form

```
class class_name {
  permission_label_1:
    member1;
  permission_label_2:
    member2;
  ...
} object_name;
```

Explain

where ***class_name*** is a name for the class (user defined *type*) and the optional field ***object_name*** is one, or several, valid object identifiers. The body of the declaration can contain ***members***, that can be either data or function declarations, and optionally ***permission labels***, that can be any of these three keywords: **private:**, **public:** or **protected:**.

Members

- **private** members of a class are accessible only from other members of their same class or from their "*friend*" classes.
- **protected** members are accessible from members of their same class and *friend* classes, and also from members of their *derived* classes.

Members

- Finally, **public** members are accessible from anywhere the class is visible.

If we declare members of a class before including any permission label, the members are considered **private**, since it is the default permission that the members of a class declared with the **class** keyword acquire.

For example

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

Explain

Declares class **CRectangle** and an object called **rect** of this class (type). This class contains four members: two variables of type **int** (**x** and **y**) in the **private** section (because private is the default permission) and two functions in the **public** section: **set_values()** and **area()**, of which we have only included the prototype.

The point

On successive instructions in the body of the program we can refer to any of the public members of the object **rect** as if they were normal functions or variables, just by putting the object's name followed by a point and then the class member

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

The program 12-1

```
#include <iostream.h>
class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area (void) {return (x*y);}
};
```

The program 12-1

```
void CRectangle::set_values  (int a, int b)
{
  x = a;
  y = b;
}
int main () {
  CRectangle rect;
  rect.set_values  (3,4);
  cout << "area: " << rect.area();
}
```

The operator ::

The new thing in this code is the operator **::** of scope included in the definition of **set_values()**. It is used to declare a member of a class outside it. Notice that we have defined the behavior of function **area()** within the definition of the **CRectangle** class - given its extreme simplicity. Whereas **set_values()** has only its protype declared within the class but its definition is outside. In this outside declaration we must use the operator of scope **::**.

Constructors and destructors

In order to avoid that, a class can include a special function: a *constructor*, which can be declared by naming a member function with the same name as the class. This constructor *function* will be called automatically when a new instance of the class is created (when declaring a new object or allocating an object of that class) and only then.

The program 12-2

```
#include <iostream.h>


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

The program 12-2

```
CRectangle::CRectangle  (int a, int b) {
  width = a;
  height = b;
}
int main () {
  CRectangle rect (3,4);
  CRectangle rectb (5,6);
  cout << "rect area: " << rect.area() <<
endl;
  cout << "rectb area: " << rectb.area() <<
endl;
}
```

The Destructor

The **Destructor** fulfills the opposite functionality. It is automatically called when an object is released from the memory, either because its scope of existence has finished or because it is an object dynamically assigned and it is released using operator **delete**.

A tilde (~)

The destructor must have the same name as the class with a tilde (~) as prefix and it must return no value.

The use of destructors is specially suitable when an object assigns dynamic memory during its life and at the moment of being destroyed we want to release the memory that it has used.

The program 12-3

```
#include <iostream.h>
class CRectangle {
    int *width, *height;
  public:
    CRectangle (int,int);
    ~CRectangle ();
    int area (void) {return (*width *
*height);}
};
```

The program 12-3

```
CRectangle::CRectangle  (int a, int b) {
  width = new int;
  height = new int;
  *width = a;
  *height = b;
}
CRectangle::~CRectangle  () {
  delete width;
  delete height;
}
```

The program 12-3

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

Overloading Constructors

In fact, in the cases where we declare a class and we do not specify any constructor the compiler automatically assumes two overloaded constructors ("*default constructor*" and "*copy constructor*")

Overloading Constructors

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

with no constructors, the compiler automatically assumes that it has the following constructor member functions:

Empty constructor

It is a constructor with no parameters defined as ***nop*** (empty block of instructions). It does nothing.

```
CExample::CExample () { };
```

Copy constructor

It is a constructor with only one parameter of its same type that assigns to every nonstatic class member variable of the object a copy of the passed object.

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

It is important

It is important to realize that both default constructors: the *empty construction* and the *copy constructor* exist only if no other constructor is explicitly declared. In case that any constructor with any number of parameters is declared, none of these two default constructors will exist.

Pointers to classes

It is perfectly valid to create pointers pointing to classes, in order to do that we must simply consider that once declared, the class becomes a valid type, so use the *class name* as the type for the pointer.

CRectangle * prect;

For example

```
#include <iostream.h>
class CRectangle {
    int width, height;
  public:
    void set_values (int, int);
    int area (void) {return (width *
height);}
};
void CRectangle::set_values (int a, int b)
{
  width = a;
  height = b;
}
```

For example

```
int main () {
  CRectangle a, *b, *c;
  CRectangle * d = new CRectangle[2];
  b= new CRectangle;
  c= &a;
  a.set_values (1,2);
  b->set_values (3,4);
  d->set_values (5,6);
  d[1].set_values (7,8);
  cout << "a area: " << a.area() << endl;
  cout << "*b area: " << b->area() << endl;
  cout << "*c area: " << c->area() << endl;
  cout << "d[0] area: " << d[0].area() <<
endl;
  cout << "d[1] area: " << d[1].area() <<
endl;
  return 0;
}
```

For example

Next you have a summary on how can you read some pointer and class operators (***, &, ., ->, [ ]**) that appear in the previous example:

For example

***x** *can be read:* pointed by **x**

&x *can be read:* address of **x**

x.y *can be read:* member **y** of object **x**

(*x).y *can be read:* member **y** of object pointed by **x**

x->y *can be read:* member **y** of object pointed by **x**

x[0] *can be read:* first object pointed by **x**

x[1] *can be read:* second object pointed by **x**

x[n] *can be read:* $(n+1)^{th}$ object pointed by **x**

Have you questions?

Summary

- ***Class and Object***
- ***Declare a class***
- ***Constructors***
- ***Destructors***

End of…

Thank you for

ATTENTION