Object-oriented Programming
Lab 03
Constructors, Destructors and Static members
Version 1.0

We have seen, so far, a few examples of classes being implemented. In all the cases, we have used member function to provide initial values to the private member variables.

Constructors

C++ provides a special member function called constructor which enables an object to initialize itself when it is created. In object-oriented programming, a constructor (sometimes shortened to ctor) in a class is a special type of subroutine called at the creation of an object. It prepares the new object for use, often accepting parameters which the constructor uses to set any member variables required when the object is first created. It is called a constructor because it constructs the values of data members of the class.

Now we revisit our previous lab (lab 02), where we developed a program for rectangle;

```
// 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;
}
```

```
//output
rect area: 12
rectb area: 30
```

Here we create two objects of class `CRectangle` named rect and rectb. For both the objects we need to call the member function `set_values (int,int)` to initialize the length and width of each rectangle object.

Constructors have the same name of the class (thus they are identified to be constructors). They have no return value. Constructors are typically used to initialize member variables of the

class to appropriate default values, or to allow the user to easily initialize those member variables to whatever values are desired.

We are going to implement CRectangle including a constructor. Lets see the program below;

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

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

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;
  return 0;
}
```

```
//output
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 width and height 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 includes a return value; not even void.

Now, type the above program code and analyze yourself!!!

Unlike normal functions, constructors have specific rules;

1. Constructors should always have the same name as the class (with the same capitalization)
2. Constructors have no return type (not even void)
3. Invoked automatically when the objects are created.
4. Have no return type, not even void, so, cannot return value.

A constructor that takes no parameters (or has all optional parameters) is called a **default constructor**.

Types of constructors

Constructors that can take arguments are termed as parameterized constructors. The number of arguments can be greater or equal to one(1). For example:

```
class example
{
     int p, q;
   public:
     example(int a, int b);          //parameterized constructor
};
example :: example(int a, int b)
{
     p = a;
     q = b;
}
```

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly. The method of calling the constructor implicitly is also called the shorthand method.

```
  example e = example(0, 50);                    //explicit call

    example e(0, 50);                              //implicit call
```

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:

Let's see the previous example of rectangle class;

```
// 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;
}
```

```
//output
rect area: 12
rectb area: 25
```

Here, `rect (3,4)` object calls the `CRectangle (int,int);` constructor and object calls the `CRectangle ();` constructor. 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!
```

Copy constructor

A copy constructor is a special constructor in the C++ programming language for creating a new object as a copy of an existing object. The first argument of such a constructor is a reference to an object of the same type as is being constructed (const or non-const), which might be followed by parameters of any type (all having default values).

Normally the compiler automatically creates a copy constructor for each class (known as a default copy constructor) but for special cases the programmer creates the copy constructor, known as a user-defined copy constructor. In such cases, the compiler does not create one. Hence, there is always one copy constructor that is either defined by the user or by the system.

Let's see the use of copy constructor;

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

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

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

CRectangle::CRectangle (int a, int b) {
  width = a;
  height = b;
}
CRectangle::CRectangle (CRectangle & X) {
//copy constructor definition
  width = X. width;
  height = X. height;
}

int main () {
  CRectangle rect (3,4);
  CRectangle rectb(rect);// copy constructor called
  CRectangle rectc = rectb; // copy constructor called again
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  cout << "rectc area: " << rectc.area() << endl;
  return 0;
}
```

```
//output
rect area: 12
rectb area: 12
rectc area: 12
```

Here for the following two statements, the copy constructor called

```
CRectangle rectb(rect);// copy constructor called
CRectangle rectc = rectb; // copy constructor called again
```

There are 3 situations in which the copy constructor is called:

1. When we make copy of an object.
2. When we pass an object as an argument by value to a method.

3. When we return an object from a method by value.

Destructor:

A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.
Following example explain the concept of destructor:

```
// destructors
#include <iostream>
using namespace std;
class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line();   // This is the constructor declaration
      ~Line();  // This is the destructor: declaration

   private:
      double length;
};

// Member functions definitions including constructor
Line::Line(void){
    cout << "Object is being created" << endl;
}
Line::~Line(void){
    cout << "Object is being deleted" << endl;
}

void Line::setLength( double len ){
    length = len;
}

double Line::getLength( void ){
    return length;
}
// Main function for the program
int main( ){
   Line line;

   // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength()
<<endl;

   return 0;
}
```

```
//output
Object is being created
Length of line : 6
Object is being deleted
```

For simple classes, a destructor is not needed because C++ will automatically clean up the memory for you. However, if you have dynamically allocated memory, or if you need to do some kind of maintenance before the class is destroyed (eg. closing a file), the destructor is the perfect place to do so.
Like constructors, destructors have specific naming rules:

1. The destructor must have the same name as the class, preceded by a tilde (~).
2. The destructor cannot take arguments.
3. The destructor has no return type.

Static members

A class can contain static members, either data or functions.
Static data members of a class are also known as "class variables", because there is only one unique value for all the objects of that same class. Their content is not different from one object of this class to another.
For example, it may be used for a variable within a class that can contain a counter with the number of objects of that class that are currently allocated, as in the following example:

```
// static members in classes
#include <iostream>
using namespace std;

class CDummy {
  public:
    static int n;
    CDummy () { n++; };
    ~CDummy () { n--; };
};

int CDummy::n=0;

int main () {
  CDummy a;
  CDummy b[5];
  CDummy *c = new CDummy;
  cout << a.n << endl;
  delete c;
  cout << CDummy::n << endl;
  return 0;
}
```

```
//output
7
6
```

In fact, static members have the same properties as global variables but they enjoy class scope. For that reason, and to avoid them to be declared several times, we can only include the prototype (its declaration) in the class declaration but not its definition (its initialization). In order to initialize a static data-member we must include a formal definition outside the class, in the global scope, as in the previous example:

int CDummy::n=0;
Because it is a unique variable value for all the objects of the same class, it can be referred to as a member of any object of that class or even directly by the class name (of course this is only valid for static members):

```
cout << a.n;
cout << CDummy::n;
```

References:

1. http://en.wikipedia.org/wiki/Constructor_(object-oriented_programming)
2. http://www.cplusplus.com/doc/tutorial/classes/
3. http://www.tutorialspoint.com/cplusplus/cpp_constructor_destructor.htm
4. Object-oriented Programming with C++ by E Balagurusamy
5. http://www.learncpp.com