Lecture-4

Abu Saleh Musa Miah

M.Sc. Engg(On going)

University of Rajshahi

.Friend class Example:

It is possible for one class to be a friend of another class. When this is the case, the friend class and all of its member functions have access to the private members defined within the other class.

```
include <iostream>
using namespace std;
  class TwoValues {
     int a;
      Int b;
public:
TwoValues(int i, int j) { a = i; b =
j; }
 friend class Min; };
```

```
class Min {
        public:
                int min(TwoValues x);
             };
int min(TwoValues x)
     {
       return x.a < x.b ? x.a : x.b;
       }
int main()
{
TwoValues ob(10, 20);
Min m;
cout << m.min(ob);
  return 0;
}
```

Inline function:

There is an important feature in C++, called an inline function, that is commonly used with classes. you can create short functions that are not actually called; rather, their code is expanded in line at the point of each invocation. This process is similar to using a function-like macro. To cause a function to be expanded in line rather than called,precede its definition with the inline keyword.

```
#include <iostream>
using namespace std;
class myclass {
int a, b;
                public:
void init(int i, int j);
void show();
                 };
// Create an inline function.
inline void myclass::init(int i, int j)
{
a = i;
b = j;
}
```

```
inline void myclass::show()
{
cout << a << " " << b << "\n";
}

int main()
{
myclass x;
        x.init(10, 20);
        x.show();
return 0;
}
```

.Parameterized Constructor:

It is possible to pass arguments to constructor functions. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

```
 #include <iostream>
using namespace std;
class myclass {
int a, b;
public:
myclass(int i, int j) {a=i; b=j;}
void show() {cout << a << " " << b;}
                    };
int main()
{
myclass ob(3, 5);
ob.show();
return 0;
}
```

.Static Class Member:

Both function and data members of a class can be made static. This section explains the consequences of each.

Static data member:

When you precede a member variable's declaration with static, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable All static variables are initialized to zero before the first object

```
#include <iostream>
using namespace std;
class shared {
        static int a;
                int b;
 public:
   void set(int i, int j)
  {a=i;
 b=j;}

 void show();
        } ;
```

```
int shared::a; // define a
void shared::show()
{
cout << "This is static a: " << a;
cout << "\nThis is non-static b: " << b;
cout << "\n";
        }
```

.Static data member:

```
int main()
{
shared x, y;
x.set(1, 1); // set a to 1
x.show();
y.set(2, 2); // change a to
2
y.show();
x.show(); /* Here, a has
been changed for both x
and y
because a is shared by
both objects. */
return 0;
}
```

```
static a: 1
non-static b: 1
static a: 2
non-static b: 2
static a: 2
non-static b: 1
```

.Static member function:

Member functions may also be declared as static. There are several restrictions placed on static member functions. They may only directly refer to other static members of the class.

```
#include <iostream>
using namespace std;
class cl {
        static int resource;
 public:
        static int get_resourc ();
         void free_resource() { resource = 0;
}    };
int cl::resource; // define resource
int cl::get_resource()
{
            if(resource) return 0;
else {
           resource = 1;
          return 1;
    }            }
```

```
int main()
{
cl ob1, ob2;
if(cl::get_resource()) cout << "ob1 has
resource\n";
      if(!cl::get_resource()) cout << "ob2
denied resource\n";
          ob1.free_resource();
         if(ob2.get_resource())

cout << "ob2 can now use resource\n";
return 0;
}
```

When Constructor and destructor are Execution:

As a general rule, an object's constructor is called when the object comes into existence, and an object's destructor is called when the object is destroyed. Precisely when these events occur is discussed here.

```
#include <iostream>
using namespace std;
            class myclass {
                        public:
            int who;
               myclass(int id);
                  ~myclass();
            } glob_ob1(1), glob_ob2(2);
myclass::myclass(int id)
          {
                cout << "Initializing " << id <<
"\n";
               who = id;
}
 myclass::~myclass()
        {
             cout << "Destructing " << who
```

```
int main()
{
myclass local_ob1(3);
cout << "This will not be first line
displayed.\n";
myclass local_ob2(4);
return 0;
}
```

It displays this output:

```
Initializing 1
Initializing 2
Initializing 3
This will not be first line displayed.
Initializing 4
```

.The scope resolution operator:

The :: operator links a class name with a member name in order to tell the compiler what class the member belongs to. However, the scope resolution operator has another related use: it can allow access to a name in an enclosing scope that is "hidden" by a local declaration of the same name.

```
#include <iostream>
using namespace std;
            class myclass {
                        public:
            int who;
               myclass(int id);
                  ~myclass();
            } glob_ob1(1), glob_ob2(2);
myclass::myclass(int id)
          {
                cout << "Initializing " << id <<
"\n";
               who = id;
}
 myclass::~myclass()
        {
             cout << "Destructing " << who
```

```
int main()
{
myclass local_ob1(3);
cout << "This will not be first line
displayed.\n";
myclass local_ob2(4);
return 0;
}
```

It displays this output:

```
Initializing 1
Initializing 2
Initializing 3
This will not be first line displayed.
Initializing 4
```

.The scope resolution operator:

The :: operator links a class name with a member name in order to tell the compiler

- what class the member belongs to.
- the scope resolution operator has another related use: it can allow access to a name in an enclosing scope that is "hidden" by a local declaration of the same name.

.Nested class

- It is possible to define one class within another.
- Doing so creates a nested class.
- Since a class declaration does, in fact, define a scope, a nested class is valid only within the scope of the enclosing class.
- Frankly, nested classes are seldom used.

.Nested class

Member functions may also be declared as static. There are several restrictions placed on static member functions. They may only directly refer to other static members of the class.

```
#include <iostream>
using namespace std;
        void f();
int main()
        {
            f();
              // myclass not known here
return 0;
}
```

```
void f()
{
  class myclass {
          int i;
        public:
            void put_i(int n) { i=n; }
int get_i() { return i; }
} ob;
ob.put_i(10);
cout << ob.get_i();
}
```

Passing object to function

Objects may be passed to functions in just the same way that any other type of variable can. Objects are passed to functions through the use of the standard call-by value mechanism. This means that a copy of an object is made when it is passed to a function.

```
class myclass {
int i;
public:
myclass(int n);
~myclass();
void set_i(int n) { i=n; }
int get_i() { return i; }
};
```

```
myclass::myclass(int n)
{
i = n;
cout << "Constructing " << i <<
"\n";
}
myclass::~myclass()
{
cout << "Destroying " << i << "\n";
}
void f(myclass ob);
```