Object-Oriented Programming

Lesson 10

Keyword static
Namespaces

Keyword static

In C++ the keyword static has two basic meanings:

1. Allocated once at a fixed address; that is, the object is created in a special static data area rather than on the stack. This is the concept of static storage.

2. Local to a particular translation unit. Here, static controls the visibility of a name, so that name cannot be seen outside the translation unit or class. This also describes the concept of linkage, which determines what names the linker will see.

- static variables and objects
- static members (data and functions)
- controlling linkage

Static variables inside functions

When you create a local variable inside a function, the compiler allocates (on stack) storage for that variable each time the function is called.

If you want to retain a value between function calls, you declare that variable as static. The storage for this variable is not on the stack but instead in the program's static data area. This variable is initialized only once, the first time the function is called.

```
void compare1 (int a)
{
   static int max;
   if (a > max) cout << "positive" << endl;
   else         cout << "negative" << endl;
}
```

Initialization to 0
(for *built-in* types)

```
 compare1 (400);
 compare1 (-40);
```

```
positive
negative
```

Static variables inside functions

```
class A
{
    /* … */
};
```

```
void compare2 (int a)
{
    static A max;
    /* … */
}
```

Initialization requires availability of the default constructor

Static objects

1. Static objects are allocated in the program’s static data area.

2. Static objects are guaranteed to be initialized before being used.

3. User-defined types must be initialized with constructor calls. Thus, if you don’t specify constructor arguments when you define the static object, the class must have a default constructor.

4. Destructors for static objects are called when **main( )** exits or when the Standard C library function **exit( )** is explicitly called.

5. Global objects are always constructed before **main( )** is entered and destroyed as **main( )** exits.

6. If a function containing a local static object is never called, the constructor for that object is never executed, so the destructor is also not executed.

Static member functions

You can create static member functions that, like static data members, work for the class as a whole rather than for a particular object of a class.

A static member function cannot access ordinary data members, only static data members, and can only call other static member functions.

A static member has no this!

A static member function cannot be const nor virtual!

Non-static member functions have full access to all static data members.

```
class CStatic
{   static int s_nInstances;
public:
    //...
    static int GetCount();
};
```

```
int CStatic::GetCount()
{
    return s_nInstances;
}
```

```
cout << CStatic::GetCount() << endl;
CStatic s3 = s2;
cout << s2.GetCount() << endl;
```

Controlling linkage

```
static A obj;
```

Global variables and ordinary functions have external linkage (i.e. at link time a name is visible to the linker everywhere, external to the translation unit where the name was defined).

There are times when you'd like to limit the visibility of a name. You might like to have a variable at file scope so all the functions in that file can use it, but you don't want functions outside that file to see or access that variable, or to inadvertently cause name clashes with identifiers outside the file.

An object or function name at file scope that is explicitly declared static is local to its translation unit (the **cpp** file where the declaration occurs). That name has internal linkage. This means that you can use the same name in other translation units without a name clash.

Controlling linkage

```
class X
{   int m_nInt;
public:
    X();
};
```

```
int main()
{
      X x1;
      Y y1;
      return 0;
}
```

```
class Y
{  int m_nInt;
public:
   Y();
};
```

```
static int number = 5;

static void ff (int a)
{   /*…*/
};

/*…*/
```

```
static int number = 100;

static void ff (int a)
{      /*…*/
};

/*…*/
```

If you remove the static keyword, the linker will report an error!

Namespaces

The names of global functions, global variables, and classes are in a single global name space. The static keyword gives you some control over this by allowing you to give variables and functions internal linkage. But in a large project, lack of control over the global name space can cause problems.

You can subdivide the global name space into more manageable pieces using the namespace feature of C++. The namespace keyword puts the names of its members in a distinct space.

```
namespace my_globals_1
{
   int a = 3;
   A aa(10);
}
```

```
namespace my_globals_2
{
   int a = 6;
   A aa(20);
}
```

```
int main(int argc, char* argv[])
{
   cout << my_globals_1::a << endl;
   using namespace my_globals_2;
   cout << a;
   return 0;
}
```

Namespaces

A namespace definition can appear only at global scope, or nested within another namespace.

No terminating semicolon is necessary after the closing brace of a namespace definition.

A namespace definition can be “continued” over multiple header files.

A namespace name can be aliased to another name, so you don’t have to type an unwieldy name created by a library vendor.

You cannot create an instance of a namespace as you can with a class.

Namespaces

```
class X
{
  int m_nInt;
public:
  X();
};
```

```
int main()
{
  X x1;
  my::Y y1;
  return 0;
}
```

```
namespace my
{
  class Y
  {
        int m_nInt;
    public:
        Y();
  };
}
```

```
int number = 5;

void ff (int a)
{  cout << a+number;  }

X::X() : m_nInt(number)
{  ff(m_nInt);  }
```

```
namespace my
{
  int number = 100;
   void ff (int a)
   { cout << a+number;  }
}

my::Y::Y() : m_nInt(number)
{  ff(m_nInt);  }
```

Scope resolution operator

Namespaces

A namespace name can be aliased to another name:

```
namespace ui = user_interface_block;
int main()
{
        ui::Y y1;
        user_interface_block::Y y2;

        return 0;
}
```

Using a namespace

```
namespace N
{
   int x = 6; int y;
   A a1(20);  A a2(20);

   class character
   {
      char m_ch;
   public:
      character (char c)
      { m_ch = c; }
      void print () const;
   };
}

void N::character::print() const
{  cout << m_ch;  }
```

1. Scope resolution operator
2. A **using** directive
3. A **using** declaration

```
int main(void)
{
   cout << N::x << endl;
   using N::y;
   cout << y << endl;
   using namespace N;
   cout << a1.GetInt();
   character ch1('f');
}
```

Bibliography

Bruce Eckel, Thinking in C++, 2nd edition, MindView, Inc., 2003

=> Chapter 10