Appendix F: Namespaces

Introduction

Namespaces are an ANSI C++ feature that allows programmers to create a scope for global identifiers. They are useful in preventing errors when two or more global declarations use the same name.

For example, assume you are a programmer in the accounting department of a business. Your company has purchased two libraries of C++ code from a software vendor. One of the libraries is designed to handle customer accounts, and contains a class object named `payable`. The other library is designed to handle company payroll, and also has a class object named `payable`. You are writing a program that integrates both libraries, but the compiler generates an error because the two class objects have the same name. You cannot modify the libraries because the software vendor does not sell the source code, only libraries of object code.

This problem can be solved when the software vendor places each library in its own namespace. Each namespace has its own name, which must be used to qualify the name of its members. For instance, the `payable` object that is part of the customer accounts library might exist in a namespace named `customer`, while the object that is part of the employee payroll library might exist in a namespace named `payroll`. When you, the application programmer, work with the objects, you must specify the namespace that the object is a member of. One way of accomplishing this is by extending the name of the object with the namespace name, using the scope resolution operator. For example, the `payable` object that is a member of the `customer` namespace is specified as `customer::payable`, and the object that is a member of the `payroll` namespace is specified as `payroll::payable`. Another way to specify the namespace is by placing a `using namespace` statement in the source file that references the namespace's member object. For example, the following statement (placed near the beginning of a source file) instructs the compiler that the file uses members of the `customer` namespace.

```
using namespace customer;
```

Likewise, the following statement instructs the compiler that the source file uses members of the `payroll` namespace:

```
using namespace payroll;
```

When a `using namespace` statement has been placed in a source file, it is no longer necessary for statements in that source file to qualify the names of the namespace's members with the namespace name and the scope resolution operator.

Defining a Namespace

A namespace is defined in the following manner.

```
namespace namespace_name
{
   declarations…
}
```

For example, look at Program F-1. It defines the `test` namespace, which has three members: `x`, `y`, and `z`.

Program F-1

```
1  // Demonstrates a simple namespace
2  #include <iostream>
3  using namespace std;
4
5  namespace test
6  {
7     int x, y, z;
8  }
9
10 int main()
11 {
12    test::x = 10;
13    test::y = 20;
14    test::z = 30;
15    cout << "The values are:\n";
16    cout << test::x << " " << test::y
17         << " " << test::z << endl;
18    return 0;
19 }
```

Program Output

```
The values are:
10 20 30
```

In Program F-1, the variables `x`, `y`, and `z` are defined in the `test` namespace's scope. Each time the program accesses one of these variables, `test::` must precede the variable name. Otherwise, a compiler error will occur.

Program F-2 demonstrates how programmers can use namespaces to resolve naming conflicts. The program defines two namespaces, `test1` and `test2`. Both namespaces have variables named `x`, `y`, and `z` as members.

Program F-2

```
1  // Demonstrates two namespaces
2  #include <iostream>
3  using namespace std;
4
5  namespace test1
6  {
7     int x, y, z;
8  }
9
10 namespace test2
11 {
12    int x, y, z;
13 }
14
15 int main()
16 {
17    test1::x = 10;
18    test1::y = 20;
19    test1::z = 30;
20    cout << "The test1 values are:\n";
21    cout << test1::x << " " << test1::y
22         << " " << test1::z << endl;
23    test2::x = 1;
24    test2::y = 2;
25    test2::z = 3;
26    cout << "The test2 values are:\n";
27    cout << test2::x << " " << test2::y
28         << " " << test2::z << endl;
29    return 0;
30 }
```

Program Output

```
The test1 values are:
10 20 30
The test2 values are:
1 2 3
```

An alternative approach to qualifying the names of namespace members is to use the `using namespace` statement. This statement tells the compiler which namespace to search for an identifier, when the identifier cannot be found in the current scope. Program F-3 demonstrates the statement.

Program F-3

Contents of `nsdemo.h`

```
1  // This file defines a namespace named demo.
2  // In the demo namespace a class named NsDemo
3  // is declared, and an instance of the class
4  // named testObject is defined.
5
6  namespace demo
7  {
8     class NsDemo
9     {
10       public:
11          int x, y, z;
12    };
13
14    NsDemo testObject;
15 }
```

Contents of Main File, `PrF-3.cpp`

```
1  // A demonstration of the using namespace statement.
2  #include <iostream>
3  #include "nsdemo.h"
4  using namespace std;
5  using namespace demo;
6
7  int main()
8  {
9     testObject.x = 10;
10    testObject.y = 20;
11    testObject.z = 30;
12    cout << "The values are:\n"
13         << testObject.x << " "
14         << testObject.y << " "
15         << testObject.z << endl;
16    return 0;
17 }
```

Program Output

```
The values are:
10 20 30
```

The `using namespace demo;` statement eliminates the need to precede `testObject` with `demo::`. The compiler automatically uses the namespace demo to find the identifier.

ANSI C++ and the `std` Namespace

All the identifiers in the ANSI standard header files are part of the `std` namespace. In ANSI C++, `cin` and `cout` are written as `std::cin` and `std::cout`. If you do not wish to specify `std::` with `cin` or `cout` (or any of the ANSI standard identifiers), you must write the following statement in your program:

```
using namespace std;
```