

Chapter 14

Namespaces

Namespaces

We started out in CS 1A stating that identifiers were used in C++ to name things. We named a few variables and constants and that was about it. As we have progressed we have named user-defined functions, user-defined data types, classes and so on. There are so many things to name in C++ it is not unreasonable to expect that several things might have the same name. A namespace is simply a way to divide items into collections so that no two items in a given collection have the same definition.

Namespace - A collection of names which are typically class definitions, and variable and named constant declarations.

When we included directives in our programs such as

```
#include <iostream>
```

all global identifiers in the header file `iostream` became global identifiers in our program. This gave us the ability to use identifiers such as `cin` and `cout` in our programs. If a global identifier in a program is the same as a global identifier in the header file, the compiler generates an error as duplicate identifiers are not allowed. We have discussed that large applications programs typically contain collections of software written by third-party vendors that is incorporated into the particular application being developed. To help alleviate this problem of duplicate names, third-party vendors adopted the practice of beginning their global identifiers with the underscore (`_`). This is why we have said to avoid the use of the underscore as the starting character for identifiers in our programs. Problems can still occur so ANSI/ISO (American National Standard Institution / International Standard Organization) Standard C++ provides a mechanism called `namespace` to solve the problem of duplicate global identifiers.

The general form for defining a namespace is as follows

```
namespace NamespaceName  
{  
    namespace members  
}
```

The *using directive*

```
using namespace std;
```

tells the compiler that the program is using the `std` ("standard") namespace. When identifiers such as `cin` and `cout` were defined in `iostream`, it was indicated that they

were in the `std` namespace. If you wished to write your own versions of `cin` and `cout` for a particular application, you could omit the reference to the `std` namespace in your program and the compiler would not know that `cin` and `cout` defined in `iostream` existed. Only your versions of these functions would be known. To date, we have put the *using* directive as a global directive. This does not always have to be the case.

THE SCOPE OF A *USING* DIRECTIVE IS THE BLOCK IN WHICH IT APPEARS. *USING* DIRECTIVES OUTSIDE OF ALL BLOCKS APPLY TO THE ENTIRE FILE THAT FOLLOWS THE DIRECTIVE.

Sample specification file

```
#include <iostream>
#include <iomanip>
using namespace std;

namespace NmSpc1
{
    const char HELLO[22] = "\n\nHello from NmSpc1";
    const int VAL = 1;
    float DoMath(float n1, float n2);
}

namespace NmSpc2
{
    const char HELLO[22] = "\n\nHello from NmSpc2";
    const int VAL = 2;
    float DoMath(float n1, float n2);
}

float GlobalDoMath(float n1, float n2);
```

Driver program

```
#include "namespacedemo1.h"
int main()
{
    float firstNum, secNum;
    cout << "Enter 2 numbers: ";
    cin >> firstNum >> secNum;
    cout << fixed << showpoint;
    cout << setprecision(2);

    {
        // names in this block use names defined in NmSpc1, std, and the global namespace
        using namespace NmSpc1;
        cout << HELLO;
        cout << endl << VAL;
        cout << "\n" << firstNum << " + " << secNum << " = "
            << DoMath(firstNum, secNum);
    }

    {
        // names in this block use names defined in NmSpc2, std, and the global namespace
        using namespace NmSpc2;
        cout << HELLO;
        cout << endl << VAL;
        cout << "\n" << firstNum << " * " << secNum << " = "
            << DoMath(firstNum, secNum);
        cout << "\n\nCall to global from NmSpc2\n" << firstNum << " - "
            << secNum << " = " << GlobalDoMath(firstNum, secNum);
    }

    // names out here use names defined in std, and the global namespace only
    cout << "\n\nHello from main\n" << firstNum << " - " << secNum << " = "
        << GlobalDoMath(firstNum, secNum);

    return 0;
}
```

Implementation file

```
#include "namespacedemo1.h"
namespace NmSpc1
{
    float DoMath(float n1, float n2)
    {
        return n1 + n2;
    }
}

namespace NmSpc2
{
    float DoMath(float n1, float n2)
    {
        return n1 * n2;
    }
}

float GlobalDoMath(float n1, float n2)
{
    return n1 - n2;
}
```

Sample run:

Enter 2 numbers: 6 4

Hello from NmSpc1

1

6.00 + 4.00 = 10.00

Hello from NmSpc2

2

6.00 * 4.00 = 24.00

Call to global from NmSpc2

6.00 - 4.00 = 2.00

Hello from main

6.00 - 4.00 = 2.00

Suppose I wanted to use the DoMath function from NmSpc2 and the constant HELLO from NmSpc1. What would happen if I did the following?

```
{
    // names in this block use HELLO defined in NmSpc1 and DoMath from NmSpc2
    using namespace NmSpc1;
    using namespace NmSpc2;
    cout << HELLO;
    cout << endl << VAL;
    cout << "\n" << firstNum << " + " << secNum << " = "
        << DoMath(firstNum, secNum);
}
```

This would not compile because it is an _____
_____. We need to specify which
names from each namespace we want to use in this section. This is done with **using
declarations**.

```
{
    // names in this block use HELLO defined in NmSpc1 and DoMath from NmSpc2
    using NmSpc1::HELLO;
    using NmSpc2::DoMath;
    cout << HELLO;
    // cout << endl << VAL;
    cout << "\n" << firstNum << " + " << secNum << " = "
        << DoMath(firstNum, secNum);
}
```

Sample run:

Enter 2 numbers: 4 5

Hello from NmSpc1
4.00 + 5.00 = 20.00 // multiplication performed as per DoMath in NmSpc1

Hello from NmSpc2
4.00 * 5.00 = 20.00

Call to global from NmSpc2
4.00 - 5.00 = -1.00

Hello from main
4.00 - 5.00 = -1.00

```
{
// names in this block use HELLO defined in NmSp1 and DoMath from NmSp2
using NmSp1::HELLO;
using NmSp2::DoMath;
cout << HELLO;
cout << endl << VAL;
cout << "\n" << firstNum << " + " << secNum << " = "
    << DoMath(firstNum, secNum);
}
```

What would happen in this case?

Exercises

```
#include <cmath>
#include <iostream>
```

```
int main()
{
    int n1, n2;
    n1 = 25;
    n2 = sqrt(n1);
    cout << n2;
}
```

Make the necessary modifications so this program will compile and run without including using namespace std;

```
#include <cmath>
#include <iostream>
```

```
void Greeting();
```

```
int main()
{
    using namespace std;

    int n1, n2;

    n1 = 25;
    n2 = sqrt(n1);
    Greeting();
    cout << "The square root of " << n1 << " is " << n2;
}
```

```
void Greeting()
{
    cout << "Hello from Greeting\n\n";
}
```

Make any necessary modifications so this program will compile and run.

```

#include <iostream>
using namespace std;

char c = 'C';
int i = 5;

namespace NmSpc
{
    int val = 10;
    float c = 2.5f;
    int i = 15;
    void Greeting();
}

using namespace NmSpc;

int main()
{
    int c = 20;
    float pi = 3.14159;
    char ch = 'M';

}

void NmSpc::Greeting()
{
    cout << "Hello from Greeting\n\n";
}

```

Write the necessary output statements to produce the output shown below:

```

20
C
2.5
15
5
10
Hello from Greeting

```

Nesting Namespaces

```
#include <iostream>
#include <iomanip>
using namespace std;

char c = 'C';
int i = 5;

namespace NmSpc
{
    namespace NmSpc2
    {
        int newVal = 5000;
    }
    int val = 10;
    float c = 2.5f;
    int i = 15;
    void Greeting();
    void cin();
}

int main()
{
    int c = 20;
    float pi = 3.14159;
    char ch = 'M';

    cout << NmSpc::NmSpc2::newVal << endl;
    cout << c << endl;           // accesses c declared in main
    cout << ::c << endl;         // scope resolution used to indicate global c
    cout << NmSpc::c << endl;    // value of c declared in NmSpc
    cout << NmSpc::i << endl;    // value of i declared in NmSpc
    cout << ::i << endl;        // global i
    cout << NmSpc::val << endl;

    NmSpc::Greeting();
    NmSpc::cin();
}

void NmSpc::Greeting()
{
    cout << "Hello from Greeting\n\n";
}
void NmSpc::cin()
{
    cout << "Hello from useless cin\n\n";
}
```