MODULE 23
NAMESPACES

When the space becomes bigger and bigger
You have to define your own space!

My Training Period: hours

Note: Tested using Visual C++ 6.0 and VC++ .Net, Win32 empty console mode application. **g++** (GNU C++) example is given at the end of this Module.

Abilities

Able to:

- Understand, use and create the namespace.
- Understand and use namespace `std`.
- Using C Standard Library in C++ (C++ wrappers).
- Understand C++ Standard library.
- Understand the translation unit.

23.1 Namespace

- Real applications or programs consist of many source files. These files can be authored and maintained by more than one developer or programmer. Eventually, the separate files are organized and linked to produce the final application.
- Traditionally, the file organization requires that all names that aren't encapsulated within a defined namespace (such as in a function or class body, or translation unit) must share the same global namespace. Hence, multiple definitions of names or name clashes will be encountered while linking the separate modules.
- Namespace mechanism in C++ overcomes the problem of name clashes in the global scope. The namespace mechanism allows an application to be partitioned into number of subsystems. Each subsystem can define and operate within its own scope.
- The namespaces declaration identifies and assigned a unique name to a user declared namespace. This will be used to solve the name collision or conflict in large program and libraries development where there are many programmers or developer working for different program portions.
- To use C++ namespaces, there are two steps involved:

  1. To uniquely identify a namespace with the keyword **`namespace`**.
  2. To access the elements of an identified namespace by applying the **`using`** keyword.

- General form of namespace is:

```
namespace indentifier
{ namespace body }
```

- For example:

```
namespace  NewOne
{
 int p;
 long q
}
```

- `p` and `q` are normal variables but integrated within the `NewOne` namespace. In order to access this variables from the outside the namespace, we have to use the scope operator, **::**. From previous example:

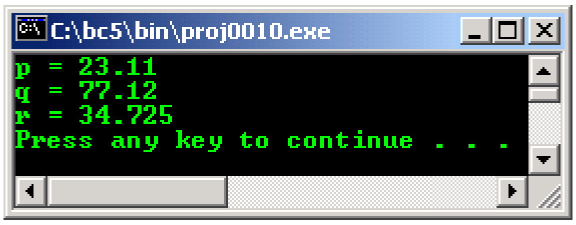
```
newOne::p;
newOne::q;
```

- A namespace definition can be nested within another namespace definition. Every namespace definition must appear either at file scope or immediately within another namespace definition. For example:

```
//namespace with using directive
```

```
#include <iostream>
#include <stdlib.h>

namespace SampleOne
{
	float p = 10.34;
}

namespace SampleTwo
{
	using namespace SampleOne;
	float q = 77.12;
	namespace InSampleTwo
	{
		float r = 34.725;
	}
}

int main()
{
	//this directive gives you everything declared in SampleTwo
	using namespace SampleTwo;
	//this directive gives you only InSampleTwo
	using namespace SampleTwo::InSampleTwo;
	//local declaration, take precedence
	float p = 23.11;

	cout<<"p = "<<p<<endl;
	cout<<"q = "<<q<<endl;
	cout<<"r = "<<r<<endl;
	system("pause");
	return 0;
}
```

Output:



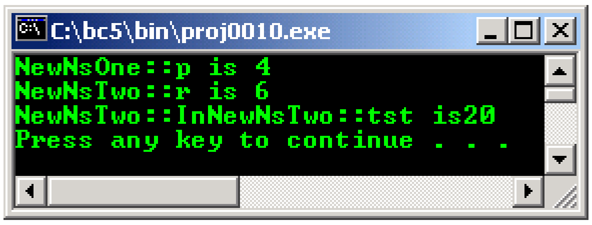
```
p = 23.11
q = 77.12
r = 34.725
Press any key to continue . . .
```

- Another program example.

```
//namespace without using directive
#include <iostream>
#include <stdlib.h>

namespace NewNsOne
{
	//declare namespace NewNsOne variable
	int p = 4;
	//declare namespace NewNsOne function
	int funct(int q);
}

namespace NewNsTwo
{
	//declare namespace NewNsTwo variable
	int r = 6;
	//declare namespace NewNsTwo function
	int funct1(int numb);
	//declare nested namespace
	namespace InNewNsTwo
	{
		//declare namespace InNewNsTwo variable
		long tst = 20.9456;
	}
}

int main()
{
	//The following four lines of code will generate error
	//because it is not at global scope...
```

```
        //namespace local
        //{
        //int k;
        //}

        cout<<"NewNsOne::p is "<<(NewNsOne::p)<<endl;
        cout<<"NewNsTwo::r is "<<(NewNsTwo::r)<<endl;

        cout<<"NewNsTwo::InNewNsTwo::tst is"<<(NewNsTwo::InNewNsTwo::tst)<<endl;
        system("pause");
        return 0;
}
```

Output:



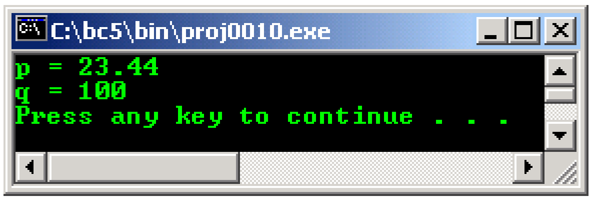
```
NewNsOne::p is 4
NewNsTwo::r is 6
NewNsTwo::InNewNsTwo::tst is20
Press any key to continue . . .
```

23.1.1 Namespace Alias

- Alternative name can be used to refer to a namespace identifier. An alias is useful when you need to simplify the long namespace identifier.
- Program example:

```
//namespace alias
#include <iostream>
#include <stdlib.h>

namespace TheFirstLongNamespaceSample
{
 float p = 23.44;
        namespace TheNestedFirstLongNamespaceSample
        {
                int q = 100;
        }
}

//Alias namespace
namespace First = TheFirstLongNamespaceSample;

//Use access qualifier to alias a nested namespace
namespace Second = TheFirstLongNamespaceSample::TheNestedFirstLongNamespaceSample;

int main()
{
        using namespace First;
        using namespace Second;

        cout<<"p = "<<p<<endl;
        cout<<"q = "<<q<<endl;

        system("pause");
        return 0;
}
```

Output:



```
p = 23.44
q = 100
Press any key to continue . . .
```

23.1.2 Namespace Extension

- The definition of a namespace can be split over several parts of a single translation unit.

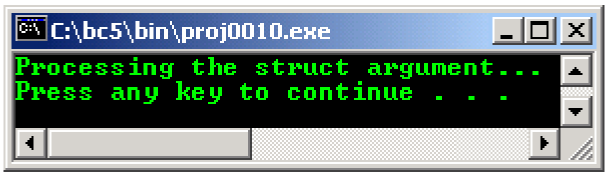
- If you have declared a namespace, you can extend the original namespace by adding new declarations.
- Any extensions that are made to a namespace after a `using` declaration will not be known at the point at which the using declaration occurs.
- For example:

```
//namespace extension
//cannot be compiled, no output, just sample code

//original namespace
namespace One
{
       //declare namespace One variable
       int p;
       int q;
}

namespace Two
{
       float r;
       int s;
}

//namespace extension of the One
namespace One
{
       //declare namespace One function
       void funct1(void);
       int funct2(int p);
}

int main()
{
}
//no output
```

- Another program example.

```
//namespace extension
#include <iostream>
#include <stdlib.h>

struct SampleOne
{
};
struct SampleTwo
{
};

//original namespace
namespace NsOne
{
       //original function...
       void FunctOne(struct SampleOne)
       {
              cout<<"Processing the struct argument..."<<endl;
       }
}

using NsOne::FunctOne;  //using-declaration...

//extending the NsOne namespace
namespace NsOne
{
       //overloaded function...
       void FunctOne(SampleTwo&)
       {
              cout<<"Processing the function argument..."<<endl;
       }
}


int main()
{
      SampleOne TestStruct;
      SampleTwo TestClass;

      FunctOne(TestStruct);
      //The following function call fails because there are
      //no overloaded version for this one
      //FunctOne(TestClass);

       system("pause");
       return 0;
}
```

Output:



```
C:\bc5\bin\proj0010.exe
Processing the struct argument...
Press any key to continue . . .
```

23.1.3 Unnamed/anonymous Namespace

- Use the keyword `namespace` without identifier before the closing brace. This can be superior alternative to the use of the **global static variable** declaration.
- Each identifier that is enclosed within an unnamed namespace is unique within the translation unit in which the unnamed namespace is defined.
- The syntax:

```
namespace { namespace_body }
```

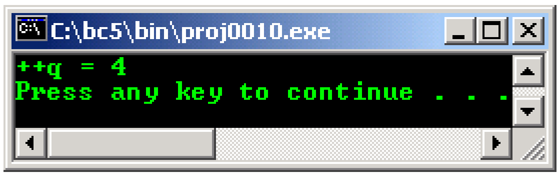
- Behaves **as if** it were replaced by:

```
namespace unique { namespace_body}
using namespace unique;
```

- Program example:

```
//anonymous or unnamed namespace
#include <iostream>
#include <stdlib.h>

//Anonymous namespace
namespace
{
       int p = 1;  //unique::p
}

void funct1()
{
       ++p;  //unique::++p
}

namespace One
{
       //Nested anonymous namespace
       namespace
       {
              int p;      //One::unique::p
              int q = 3;  //One::unique::q
       }
}

//using-declaration
using namespace One;

void testing()
{
       //++p;   // error, unique::p or One::unique::p?
       //One::++p;   //error, One::p is undefined
       cout<<"++q = "<<++q<<endl;
}

int main()
{
       testing();
       system("pause");
       return 0;
}
```

Output:


C:\bc5\bin\proj0010.exe

```
++q = 4
Press any key to continue . . .
```

23.2 Accessing Namespace Elements

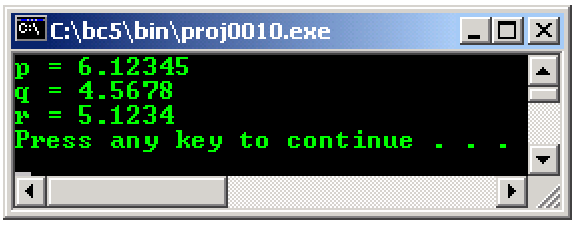
- There are several methods to access namespace elements as listed below:
  - Using explicit access qualification
  - Using directive
  - Using declaration

23.2.1 using Directive

- This method is useful when you wan to access several or all the members of a namespace.
- This `using`-directive specifies that all identifiers in a namespace are in scope at the point that the `using`-directive statement is made.
- It is also transitive; this means that when you apply the `using` directive to a namespace that contains `using` directive within itself, you get access to those namespaces as well.
- Program example:

```
//using directive
#include <iostream>
#include <stdlib.h>

namespace One
{
       float p = 3.1234;
}

namespace Two
{
       using namespace One;

              float q = 4.5678;
              namespace InTwo
              {
                     float r = 5.1234;
              }
}

int main()
{

       //This using directive gives you all declared in Two
       using namespace Two;
       //This using directive gives you only the InTwo namespace
       using namespace Two::InTwo;
       //This is local declaration, it takes precedence
       //comment this code and re run this program, see the different...
       float p = 6.12345;

       cout<<"p = "<<p<<endl;
       cout<<"q = "<<q<<endl;
       cout<<"r = "<<r<<endl;

       system("pause");
       return 0;
}
```

Output:



```
p = 6.12345
q = 4.5678
r = 5.1234
Press any key to continue . . .
```

23.2.2 `using` Declaration

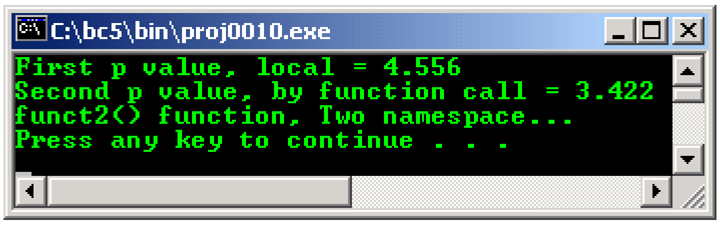
- You can access namespace elements individually by applying the using-declaration.
- Here, you add the declared identifier to the local namespace. The syntax is:

```
using::unqualified-identifier;
```

- Program example:

```
//using declaration
//Function funct2() defined in two different namespaces
#include <iostream>
#include <stdlib.h>

namespace One
{
       float funct1(float q)
       {
              return q;
       }

   //First funct2() definition
   void funct2()
   {
       cout<<"funct2() function, One namespace..."<<endl;
   }
 }
namespace Two
{
       //Second funct2() definition
       void funct2()
       {
              cout<<"funct2() function, Two namespace..."<<endl;
       }
}

int main()
{
       //The using declaration identifies the desired version of funct2()
       using One::funct1;  //Becomes qualified identifier
       using Two::funct2;  //Becomes qualified identifier
       float p = 4.556;    //Local declaration, takes precedence

       cout<<"First p value, local = "<<p<<endl;

       p = funct1(3.422);
       cout<<"Second p value, by function call = "<<p<<endl;

       funct2();

       system("pause");
       return 0;
}
```

Output:



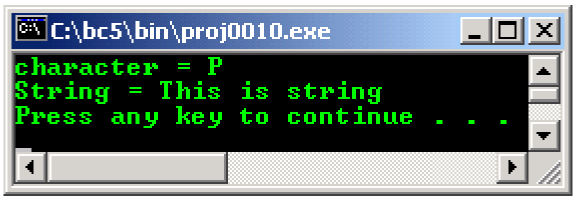
```
First p value, local = 4.556
Second p value, by function call = 3.422
funct2() function, Two namespace...
Press any key to continue . . .
```

23.2.3 Explicit Element Access

- This access method we use the namespace identifier together with the (**::**) scope resolution operator followed by the element name.
- Using this method, we can qualify each member of a name space. It also can resolve the ambiguity.
- No matter which namespace (except anonymous/unnamed namespace) is being used in your subsystem or program, you can apply the **scope operator**, **::** to access identifiers in any namespace (including a namespace already being used in the local scope) or the global namespace.
- `using` directive cannot be used inside a class but the using declaration is allowed.
- Program example:

```
//Explicit access, namespace within a class
#include <iostream>
#include <stdlib.h>

class One
{
       public:
       void funct1(char chs)
       {cout<<"character = "<<chs<<endl;}
};

class Two:public One
{
       public:
       //The using directive is not allowed in class
       //using namespace One;
       void funct1(char *str)
       {cout<<"String = "<<str<<endl;}

       //using declaration is OK in class
       using One::funct1;  //overload Two::funct1()
};

int main()
{
   Two Sample;

 //Calling One::funct1()
   Sample.funct1('P');
   //Calling Two::funct1()
   Sample.funct1("This is string");

   system("pause");
   return 0;
}
```

Output:



```
character = P
String = This is string
Press any key to continue . . .
```

23.3 Namespace std

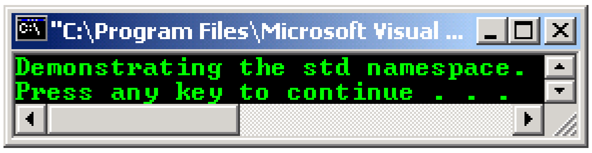
- All the classes, objects and functions of the standard C++ library are defined within namespace `std`, defined by the ISO/ANSI C++ or a new one, **ISO/IEC C++**.
- Even though compiler that comply with ISO/ANSI C++ standard allow the use of the traditional header files (such as `iostream.h`, `stdlib.h` or other than `.h` extension for implementation dependent etc), actually the standard has specified new names for these header files, using the same name but without the `.h` (or other extension for implementation dependent) under the namespace `std`. For example, `iostream.h` becomes `iostream`.
- All functions, classes and objects will be declared under the `std` namespace if using the ISO/ANSI C++ compliance compiler. The `.h` header files have been provided for backward compatibility.
- The `ISO/ANSI C++` standard requires you to **explicitly** declare the namespace in the standard library. For example, when using header file `iostream.h`, you **do not have** to specify the namespace of `cout` in one of the following ways (as used throughout this tutorial):

- `std::cout` – explicitly
- `using std::cout` – `using` declaration
- `using namespace std` – using directive

- If you use `iostream` without the `.h` extension, then you have to explicitly include either one of those three codes.
- The following is a simple program example and make sure your compiler is ISO/ANSI C++ compliance.

```
//namespace std example
//notice the omitted .h header files
#include <iostream>
#include <cstdlib>

void main()
{
   std::cout<<"Demonstrating ";
   using namespace std;
   cout<<"the std namespace."<<endl;
   system("pause");
}
```

Output:



```
Demonstrating the std namespace.
Press any key to continue . . .
```

- Then, try comment out the following line, recompile the program. Your compiler should generate error.

```
using namespace std;
```

23.4 Standard C++ Library

- C++ library entities such as functions and classes are declared or/and defined in one or more standard headers. To make use of a library entity in a program, as in C program, we have to include them using the `include` preprocessor directive, `#include`.
- The Standard C++ library headers as shown in Table 23.1 together with the 16 Standard C headers (C++ wrappers - get the ideas of the wrappers HERE) shown in Table 23.2, constitute an implementation of the C++ library.
- There are several headers that are rarely used are not included here, please check your compiler documentation.
- These headers are template based. You will learn Template in next Module (Module 24).

<algorithm>	<bitset>	<complex>
<deque>	<exception>	<fstream>
<functional>	<hash_map>	<hash_set>
<iomanip>	<ios>	<iosfwd>
<iostream>	<istream>	<iterator>
<limits>	<list>	<locale>
<map>	<memory>	<new>
<numeric>	<ostream>	<queue>
<set>	<sstream>	<stack>
<stdexcept>	<streambuf>	<string>
<strstream>	<utility>	<valarrary>
<vector>		

Table 23.1: C++ Standard header

<cassert>	<cctype>	<cerrno>
<cfloat>	<ciso646>	<climits>
<clocale>	<cmath>	<csetjmp>

<csignal>	<cstdarg>	<cstddef>
<cstdio>	<cstdlib>	<cstring>
<ctime>		

Table 23.2: C++ wrapper – Using C library in C++ codes. Note that there is no .h anymore

- If you want to use functions, structure, macros and other built-in item available in the Standard C headers in C++ programming environment or ISO/IEC C++ compilers, use the C++ wrappers.
- These C++ wrappers are C headers that prefixed by `c` character such as <`cassert`> from <`assert.h`>.
- Other than those ISO/IEC C and ISO/IEC C++ Standard headers should be implementation dependant, it is non standard. Keep in mind that there are a lot more non standard headers that you will find.
- If you want to use for example, the member functions or classes of these non standard headers, you have to get the information through their documentation.
- The issue of using the non standard library is the program portability. There is a lot of C/C++ implementation out there, so you decide it, depend on your needs and what is the target platform your program are developed for and what compiler you are going to use to develop the programs.

23.5 Using C++ Library Headers

- In C++, you include the contents of a standard header by using the include preprocessor directive as shown below:

```
#include <iostream>
#include <cstdlib>
```

- You can include the standard headers in any order, a standard header more than once, or two or more standard headers that define the same macro or the same type. Do not include a standard header within a declaration.
- A Standard C header never includes another standard header. Every function in the library is declared in a standard header.
- Unlike in Standard C, the standard C++ header never provides a masking macro, with the same name as the function that masks the function declaration and achieves the same effect.
- All names other than **operator `delete`** and **operator `new`** in the C++ library headers are defined in the `std` namespace, or in a namespace nested within the `std` namespace. You refer to the name `cout`, for example, as `std::cout`.
- The most portable way to deal with namespaces may be to obey the following two rules:

  - To assuredly declare an external name in namespace `std` that is traditionally declared in <`stdlib.h`>, for example, include the header <`cstdlib`>. Knowing that the name might also be declared in the global namespace.
  - To assuredly declare in the global namespace an external name that is declared in <`stdlib.h`>, include the header <`stdlib.h`> directly. Knowing that the name might also be declared in namespace `std`.

- For example if you want to use `std::cout`, you should include <`iostream`>. If you want to use `printf()`, you should include <`stdio.h`> instead of <`cstdio`>.
- Normally programmers use the `using` declaration:

```
using namespace std;
```

- This brings all library names into the current namespace. If you write this declaration immediately after all the `include` preprocessor directives, you hoist the names into the global namespace.
- You can subsequently ignore namespace considerations in the remainder of the translation unit. You also avoid most dialect differences across different translation environments.
- Unless specifically indicated otherwise, you may not define names in the `std` namespace, or in a namespace nested within the `std` namespace.
- The term translation unit refers to a source code file together with any included files, but less any source lines omitted by conditional preprocessor directives. Syntactically, a translation unit is defined as a sequence of external declarations:

  external-declaration

translation-unit external-declaration

external-declaration

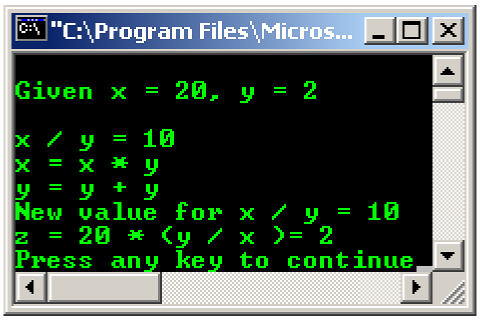
function-definition
declaration

Some Examples and Experiments

- The following examples tested by using Visual C++ 6.0 and Visual Studio .Net.
- It just a re-compilation of the program examples from other Modules, assuming that the compiler is fully ISO/IEC C++ compliance. Notice some of the code modifications.

```
/*Simple mathematics calculation*/
//This program is from module 1, C program.
//header files used is the C++ wrapper, no .h anymore.
//The stdlib.h for system("pause") also has been removed
#include <cstdio>

//main() function
int main( )
{
        //variables declaration and initialization
        int     x, y, z;
        x = 20;
        y = 2;

        printf("\nGiven x = 20, y = 2\n");
        printf("\nx / y = %d", x / y);
        x = x * y;
        y = y + y;
        printf("\nx = x * y");
        printf("\ny = y + y");
        printf("\nNew value for x / y = %d", x / y);
        z = 20 * y / x;
        printf("\nz = 20 * (y / x )= %d\n", z);

        return 0;
}
```

Output:

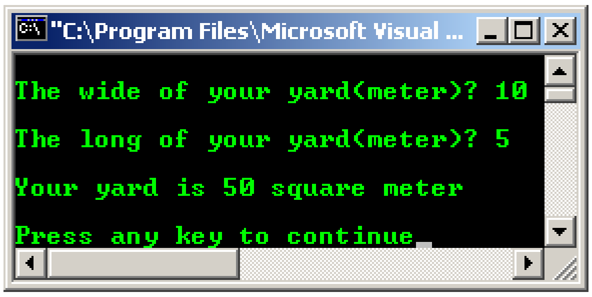


```

Given x = 20, y = 2

x / y = 10
x = x * y
y = y + y
New value for x / y = 10
z = 20 * (y / x )= 2
Press any key to continue
```

```
//demonstrates the use of function prototypes
//C++ program, no .h anymore
#include <iostream>
//but has to explicitly use the 'using namespace std'
using namespace std;

typedef unsigned short USHORT;
//another method simplifying type identifier

USHORT FindTheArea(USHORT length, USHORT width);
//function prototype

int main()
{
        USHORT lengthOfYard;
        USHORT widthOfYard;
        USHORT areaOfYard;

        cout<< "\nThe wide of your yard(meter)? ";
        cin>> widthOfYard;
```

```
       cout<< "\nThe long of your yard(meter)? ";
       cin>> lengthOfYard;

       areaOfYard = FindTheArea(lengthOfYard, widthOfYard);

       cout<< "\nYour yard is ";
       cout<< areaOfYard;
       cout<< " square meter\n\n";

       return 0;
}

USHORT FindTheArea(USHORT l, USHORT w)
{
   return (l * w);
}
```

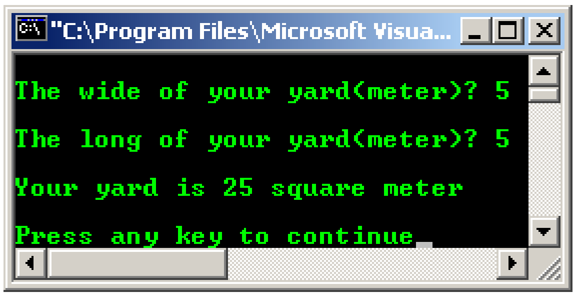
Output:



```
The wide of your yard(meter)? 10

The long of your yard(meter)? 5

Your yard is 50 square meter

Press any key to continue
```

```
//demonstrates the use of function prototypes
//variation of the C++ program, no .h anymore
//without the 'using namespace std;'
#include <iostream>

typedef unsigned short USHORT;
//another method simplifying type identifier

USHORT FindTheArea(USHORT length, USHORT width);
//function prototype

int main()
{
       USHORT lengthOfYard;
       USHORT widthOfYard;
       USHORT areaOfYard;

       //without using namespace std globally, you have to
       //explicitly use the std for every occurrences of the...
       std::cout<< "\nThe wide of your yard(meter)? ";
       std::cin>> widthOfYard;
       std::cout<< "\nThe long of your yard(meter)? ";
       std::cin>> lengthOfYard;

       areaOfYard = FindTheArea(lengthOfYard, widthOfYard);

       std::cout<< "\nYour yard is ";
       std::cout<< areaOfYard;
       std::cout<< " square meter\n\n";

       return 0;
}

USHORT FindTheArea(USHORT l, USHORT w)
{
   return (l * w);
}
```

Output:



```
The wide of your yard(meter)? 5
The long of your yard(meter)? 5
Your yard is 25 square meter
Press any key to continue
```

- Example compiled using **g++**.

```
//**************namespace.cpp*************/
//demonstrates the use of function prototypes
//variation of the C++ program, no .h anymore
//without the 'using namespace std;'
#include <iostream>

typedef unsigned short USHORT;
//another method simplifying type identifier

USHORT FindTheArea(USHORT length, USHORT width);
//function prototype

int main()
{
       USHORT lengthOfYard;
       USHORT widthOfYard;
       USHORT areaOfYard;

       //without using namespace std globally, you have to
       //explicitly use the std for every occurrences of the...
       std::cout<< "\nThe wide of your yard(meter)? ";
       std::cin>> widthOfYard;
       std::cout<< "\nThe long of your yard(meter)? ";
       std::cin>> lengthOfYard;

       areaOfYard = FindTheArea(lengthOfYard, widthOfYard);

       std::cout<< "\nYour yard is ";
       std::cout<< areaOfYard;
       std::cout<< " square meter\n\n";

       return 0;
}

USHORT FindTheArea(USHORT l, USHORT w)
{
   return (l * w);
}
```

```
[bodo@bakawali ~]$ g++ namespace.cpp -o namespace
[bodo@bakawali ~]$ ./namespace

The wide of your yard(meter)? 200

The long of your yard(meter)? 10

Your yard is 2000 square meter
```

---o0o---

Further reading and digging:

1. Check the best selling C / C++ books at Amazon.com.
2. Standards: The C / C++ standards references (ISO/IEC is covering ANSI and is more general):

   a. ISO/IEC 9899 (ISO/IEC 9899:1999) - C Programming languages.
   b. ISO/IEC 9945:2002 POSIX standard.
   c. ISO/IEC 14882:1998 on the programming language C++.
   d. ISO/IEC 9945:2003, The Single UNIX Specification, Version 3.
   e. Get the GNU C library information here.

f. Read online the GNU C library here.