Header files in C++	Namespaces in C++	using in C++	Namespace std

Header files in C++

Code files (with a .cpp extension) are not the only files commonly seen in programs. The other type of file is called a **header file**, sometimes known as an **include file**. Header files almost always have a .h extension. The purpose of a header file is to hold **declarations** for other files to use.

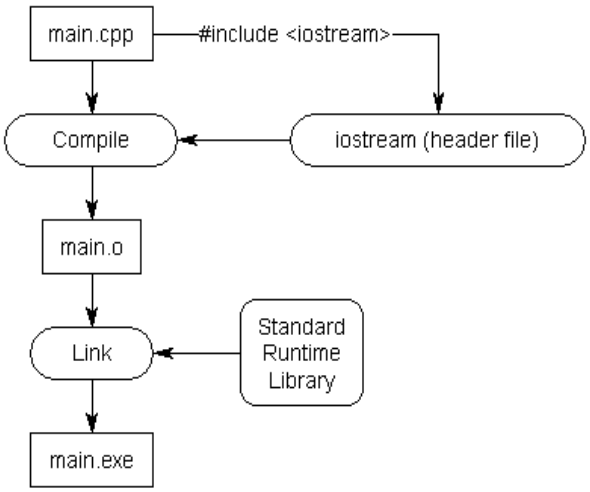
```
#include <iostream>
int main()
{
   using namespace std;
   cout << "Hello, world!" << endl;
   return 0;
}
```

The interface of C++ standard library is defined by a collection of header files e.g. **<cstdlib>**, **<exception>**, **<string>**, **<list>**, **<random>**, **<iostream>**, **<iomanip>**, **<locale>** **etc.**

Here we are using standard library header files <iostream>

This program never defines **cout**, so how does the compiler know what cout is? The answer is that cout has been **declared** in a header file called "**iostream**". When we use the line #include <iostream>, we are telling the compiler to locate and then read all the declarations from a header file named "iostream".

Keep in mind that header files **typically only contain declarations** no **definition** (implemention) and you already know that your program won't link if it can't find the implementation of something you use. So if cout is only *declared* in the "iostream" header file, where is it actually implemented ? It is implemented in the runtime support library, which is automatically linked into your program during the link phase.



- A **library** is a package of code that is meant to be reused in many programs.
- Typically, a library includes :
  - a header file , contains declarations for everything the library wishes to expose (make public) to users
  - a precompiled object that contains all of the implementation code compiled into machine language. These libraries typically have a .lib or .dll extension on Windows, and a .a or .so extension on Unix.

Writing your own header files

add.cpp:	**main.cpp:**
int add(int x, int y) { return x + y; }	#include <iostream> int add(int x, int y); // **forward declaration** using function prototype int main()

```
{
    using namespace std;
    cout << "The sum of 3 and 4 is " << add(3, 4) << endl;
    return 0;
}
```

We'd used a forward declaration so that the compiler would know what add was when compiling main.cpp. But writing forward declarations for every function you want to use that lives in another file can be a tedious task.

Header files can relieve us of this burden. The actual content of the .h file for add is shown below :

add.h:

```
#ifndef ADD_H
#define ADD_H
// function prototype for add.h
int add(int x, int y);

#endif
```

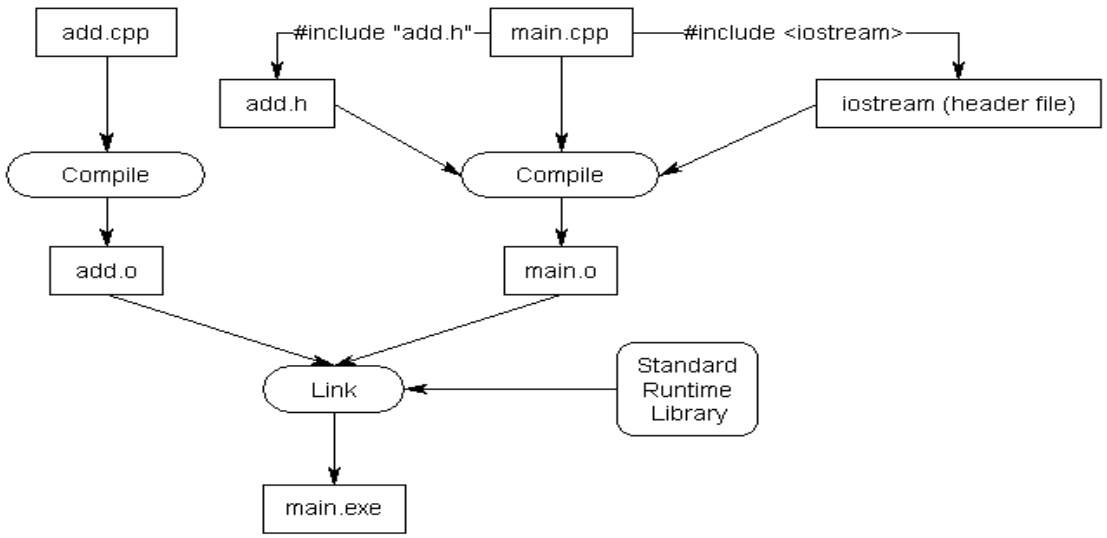
Rule: Use angled brackets to include header files that come with the compiler. Use double quotes to include any other header files.

In order to use this header file in main.cpp, we have to include it. Here is the new **main.cpp** that includes add.h:

```
#include <iostream>
// this brings in the declaration for add()
#include "add.h"

int main()
{
    using namespace std;
    cout << "The sum of 3 and 4 is " << add(3, 4) << endl;
    return 0;
}
```

When the compiler compiles the #include "add.h" line, it copies the contents of add.h into the current file. Because our add.h contains a function prototype for add(), this prototype is now being used as a forward declaration of add()! Consequently, our program will compile and link correctly.



"Why doesn't iostream have a .h extension?".

- When C++ was first created, all of the files in the standard runtime library ended in .h. The original version of cout and cin lived in iostream.h.

- When the language was standardized by the ANSI committee, they decided to move all of the functions in the runtime library into the **std namespace**. If they moved all the functions into the std namespace, none of the old programs would work any more!

- To try to get around this issue and provide backwards compatibility for older programs, a new set of header files was introduced that use the same names but lack the .h extension. These new header files have all their functionality inside the std namespace. This way, older programs that include #include <iosteam.h> do not need to be rewritten, and newer programs can #include <iostream>.

Make sure when you include a header file from the standard library that you use the **non .h version** if it exists. Otherwise you will be using a deprecated version of the header that is no longer supported.

Rule: use the non .h version of a library if it exists, and access the functionality through the std namespace. If the non .h version does not exist, or you are creating your own headers, use the .h version

Namespaces in C++

Let's start with a problem to explain what namespaces are.

- We all know that functions, classes or any other kind of identifiers can't have the same name.
- Let's say that we have two libraries where we have added a function, let's say print(). This print() function may be different in two libraries but by name they are indistinguishable. That's where namespaces come in.
- A namespace is like adding a new group name to which you can add functions or other identifiers so that they will become distinguishable.

```
namespace bla
{
 void print() {   // function bla::print() }
}
namespace blabla
{
 void print()  {   // function blabla::print() }
}
```

We can see, there are two functions with the name print, yet there is no naming conflict, because of namespaces.

So we can define namespace as

- A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it.
- Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when our code base includes multiple libraries.
- All identifiers at namespace scope are visible to one another without qualification.
- Identifiers outside the namespace can access the members by using the fully qualified name for each identifier.
- A means of grouping logically related identifiers into corresponding namespaces, thereby making the system more modular

Namespaces are like different towns or cities. You can have a street called "Main Street" in many different towns, and each "Main Street" is qualified by it's own town. For example,

Detroit::Main Street. Dallas::Main Street Bakersfield::Main Street

If you are living in Bakersfield, then you can refer to Main Street by itself, because you are already in Bakersfield so that Main Street is assumed. But if you are in Bakersfield and you are referring to the Main Street in Detroit, then you must use the namespace "Detroit" to indicate that: Detroit::Main Street.

The following example shows a namespace declaration and three ways that code outside the namespace can accesses their members.

```
namespace ContosoData
{
   class ObjectManager
   {
      public:  void DoSomething() {}
   };
   void Func(ObjectManager) {}
}
```

In order to access these identifier from outside the ContosoData namespace we have to use the **scope operator ::**

- **Use the fully qualified name:**

```
ContosoData::ObjectManager mgr;
mgr.DoSomething();          ContosoData::Func(mgr);
```

- Use a **using** declaration to bring one identifier into scope:

```
using ContosoData::ObjectManager;
ObjectManager mgr;          mgr.DoSomething();
```

- Use a **using** directive to bring everything in the namespace into scope:

```
using namespace ContosoData;

ObjectManager mgr;
mgr.DoSomething();  Func(mgr);
```

using directives

The using directive allows all the names in a **namespace** to be used without the *namespace-name* as an explicit qualifier.

- Use a using directive in an implementation file (i.e. *.cpp) if you are using several different identifiers in a namespace; if you are just using one or two identifiers, then consider a using declaration to only bring those identifiers into scope and not all the identifiers in the namespace.
- If a local variable has the same name as a namespace variable, the namespace variable is **hidden**. It is an error to have a namespace variable with the same name as a global variable.
- A using directive can be placed at the top of a .cpp file (at file scope), or inside a class or function definition.
- In general, avoid putting using directives in header files (*.h) because any file that includes that header will bring everything in the namespace into scope, which can cause name hiding and name collision problems that are very difficult to debug.
- Always use fully qualified names in a header file.

Typically, you declare a namespace in a header file. If your function implementations are in a separate file, then qualify the function names, as in this example.

```
//contosoData.h
#pragma once
namespace ContosoDataServer
{
   void Foo();
   int Bar();
}
```

Function implementations in contosodata.cpp should use the fully qualified name, even if you place a using directive at the top of the file:

```
#include "contosodata.h"
using namespace ContosoDataServer;
void ContosoDataServer::Foo() // use fully-qualified name here
{
   // no qualification needed for Bar()
   Bar();
}
```

- A namespace can be declared in **multiple blocks** in a single file, and in multiple files. The compiler joins the parts together during preprocessing and the resulting namespace contains all the members declared in all the parts. An example of this is the std namespace which is declared in each of the header files in the standard library.

Members of a named namespace can be defined outside the namespace in which they are declared by explicit qualification of the name being defined. However, the definition must appear after the point of declaration in a namespace that encloses the declaration's namespace. For example:

```
// defining_namespace_members.cpp
// C2039 expected
namespace V {     void f();     }

   void V::f() { }       // ok
```

```
  void V::g() { }      // C2039, g() is not yet a member of V

  namespace V {   void g();    }
}
```

This error can occur when namespace members are declared across multiple header files, and you have not included those headers in the correct order.

Example 1 :

```
#include <iostream>
using namespace std;
namespace first  {
    int x = 5;      int y = 10;
}
namespace second  {
    double x = 3.1416;      double y = 2.7183;    }
int main () {
    using first::x;
    using second::y;
    cout << x << endl;
    cout << y << endl;
    cout << first::y << endl;
    cout << second::x << endl;
    return 0;
}
```

Output :

```
5
2.7183
10
3.1416
```

Notice how in this code, x (without any name qualifier) refers to first::x whereas y refers to second::y, exactly as our using declarations have specified. We still have access to first::y and second::x using their fully qualified names.

The keyword using can also be used as a directive to introduce an entire namespace and then main will be :

```
int main () {
    using namespace first;
    cout << x << endl;              cout << y << endl;
    cout << second::x << endl;
    cout << second::y << endl;
    return 0;
}
```

Result : 5 10 3.1416 2.7183

Since we have declared only using namespace first, all direct uses of x and y without name qualifiers were referring to their declarations in namespace first.

using and **using namespace** have validity only in the **same block** in which they are stated or in the entire code if they are used directly in the global scope. For example, if we had the intention to first use the objects of one namespace and then those of another one, we could do something like:

```
// using namespace example
#include <iostream>
using namespace std;
namespace first
{
    int x = 5;
}
namespace second
{
    double x = 3.1416;
```

```
int main () {
  {   using namespace first;       cout << x << endl;  }
  {
    using namespace second;    cout << x << endl;
  }
  return 0;
}
Result :  5   3.1416
```

}

Example 2: Access two global variables with the same name: var.

The functionality of namespaces is especially useful in the case that there is a possibility that a global object or function uses the same identifier as another one, causing redefinition errors. For example:

```
// namespaces
#include <iostream>
using namespace std;
namespace first
{
   int var = 5;
}
namespace second
{
   double var = 3.1416;
}
```

```
int main () {
   cout << first::var << endl;
   cout << second::var << endl;
   return 0;
}
5
3.1416
```

One is defined within the namespace first and the other one in second. No redefinition errors happen because of namespaces.

Nested namespaces

```
namespace ContosoDataServer
{
  void Foo();

  namespace Details
  {
    int CountImpl;
    void Ban() { return Foo(); }
  }

  int Bar(){...};
  int Baz(int i) { return Details::CountImpl; }

}
```

- Namespaces may be nested.
- An ordinary nested namespace has unqualified access to its parent's members, but the parent members do not have unqualified access to the nested namespace (unless it is declared as inline), as shown in this example.
- The **std** namespace - All C++ standard library types and functions are declared in the std namespace or namespaces nested inside std.

Namespace aliases

Namespace names need to be unique, which means that often they should not be too short. If the length of a name makes code difficult to read, or is tedious to type in a header file where using directives can't be used, then you can make a namespace alias which serves as an abbreviation for the actual name. For example:

```
namespace a_very_long_namespace_name { class Foo {}; }
namespace AVLNN = a_very_long_namespace_name;
```

```
void Bar(AVLNN::Foo foo){ }
```

The global namespace

- If an identifier is not declared in an explicit namespace, it is part of the implicit global namespace.
- In general, try to avoid making declarations at global scope when possible, except for the entry point **main** Function, which is required to be in the global namespace.
- To explicitly qualify a global identifier, use the scope resolution operator with no name, as in ::SomeFunction(x);. This will differentiate the identifier from anything with the same name in any other namespace, and it will also help to make your code easier for others to understand.

Anonymous or Unnamed Namespaces

- 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.
- Unnamed namespace limits access of class, variable, function and objects to the file in which it is defined. All code in the same file can see the identifiers in an unnamed namespace but the identifiers, along with the namespace itself, are not visible outside that file—or more precisely outside the translation unit.
- Unnamed namespace functionality is similar to **static** keyword in C/C++. static keyword limits access of global variable and function to the file in which they are defined.
- Example :

```
namespace
{
  int MyFunc(){}
}
```

- Behaves as if it were replaced by compiler :
  - namespace *unique* { namespace_body}
  - using namespace *unique*;

- All anonymous namespaces in the same file are treated as the same namespace and all anonymous namespaces in different files are distinct.

Some important concepts and rules

- A **translation unit** refers to an implementation (.c/.cpp) file and all header (.h/.hpp) files it includes.
- If an object or function inside such a translation unit has **internal linkage**, then that specific symbol is only visible to the linker within that translation unit.
- If an object or function has **external linkage**, the linker can also see it when processing other translation units.
- The **static** keyword, when used in the global namespace, forces a symbol to have internal linkage.
- The **extern** keyword results in a symbol having external linkage.

- The compiler defaults the linkage of symbols such that:
  - Non-const global variables have external linkage by default
  - Const global variables have internal linkage by default
  - Functions have external linkage by default

Example of anonymous or unnamed namespace

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

// 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();      return 0;   }
```

The following example illustrates an improper use of unnamed namespaces.

```
#include <iostream>
using namespace std;
namespace {   const int i = 4;   }
int i = 2;
int main()
{
    cout << i << endl; // error
    return 0;
}
```

Inside main error occur because the compiler cannot distinguish between the global name and the unnamed namespace member with the same name (i). To work without error the namespace must be uniquely identified with an identifier and i must specify the namespace it is using.

Output

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

Note: Items defined in an unnamed namespace have internal linkage. Rather than using the keyword **static** to define items with internal linkage, define them in an unnamed namespace instead.

Namespace std

- std is an abbreviation of standard. std is the standard namespace. cout, cin and a lot of other things are defined in it. (This means that one way to call them is by using std::cout and std::cin.)
- The keyword using technically means, use this whenever you can. This refers, in this case, to the std namespace. So whenever the computer comes across cout, cin, endl or anything of that matter, it will read it as std::cout, std::cin or std::endl.

- All the files in the C++ standard library declare all of its entities within the std namespace. That is why we have generally included the using namespace std; statement in all programs that used any entity defined in iostream.

- Header file iostream contains the declaration part of all standard functions and it is included in the source using a **#include line.** It tells the compiler to look for the given file (header) and copy its content in place of the #include line. And, including a header file does not tell the compiler that any particular function or class or object will be found there. Basically, everything you include, plus every other header those included headers include makes up the sum-total of all the code that the compiler sees at one time (called a "translation unit") when compiling your code. That translation unit can be very large and contain code in many different namespaces (and sub-namespaces).

- Actually, the header files only contain the **declarations** of the functions... and the compiler will not know where its **actual definition** lies(i.e how this function should work). Actual implementations (definition) of functions in standard library are logically grouped under a namespace **std (no physical module or package**). In fact in the standard headers the declarations are also wrapped in the std namespace.