

C++ Scope Resolution Operator ::

C++The :: (scope resolution) operator is used to qualify hidden names so that you can still use them. You can use the unary scope operator if a namespace scope or global scope name is hidden by an explicit declaration of the same name in a block or class. For example:

```
int count = 0;

int main(void) {
    int count = 0;
    ::count = 1;    // set global count to 1
    count = 2;      // set local count to 2
    return 0;
}
```

The declaration of `count` declared in the `main()` function hides the integer named `count` declared in global namespace scope. The statement `::count = 1` accesses the variable named `count` declared in global namespace scope.

You can also use the class scope operator to qualify class names or class member names. If a class member name is hidden, you can use it by qualifying it with its class name and the class scope operator.

In the following example, the declaration of the variable `X` hides the class type `X`, but you can still use the static class member `count` by qualifying it with the class type `X` and the scope resolution operator.

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

class X
{
public:
    static int count;
};
int X::count = 10;           // define static data
member
```

```

int main ()
{
    int X = 0;           // hides class type X
    cout << X::count << endl; // use static member
of class X
}

```

Namespaces

Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name.

The format of namespaces is:

```

namespace identifier
{
    entities
}

```

Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace. For example:

```

namespace myNamespace
{
    int a, b;
}

```

In this case, the variables `a` and `b` are normal variables declared within a namespace called `myNamespace`. In order to access these variables from outside the `myNamespace` namespace we have to use the scope operator `::`. For example, to access the previous variables from outside `[myNamespace=]` we can write:

```

general::a
general::b

```

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;
}
```

EXAMPLE OUTPUT:

```
5
3.1416
```

In this case, there are two global variables with the same name: `var`. One is defined as an `int` within the namespace `first` and the other one is defined as a `double` in the namespace called `second`. No redefinition errors happen thanks to namespaces.

The 'using' Keyword

The keyword `using` is used to introduce a name from a namespace into the current declarative region. For example:

```
// using example
#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;
}

```

EXAMPLE OUTPUT:

```

5
2.7183
10
2.7183

```

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:

```

// using example
#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 namespace first;
    cout << x << endl;
    cout << y << endl;
    cout << second::y << endl;
    cout << second::x << endl;
    return 0;
}

```

EXAMPLE OUTPUT:

```

5
10
3.1416
2.7183

```

In this case, since we have declared that we were 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;
}

```

EXAMPLE OUTPUT:

```

5
3.1416

```

Namespace alias

We can declare alternate names for existing namespaces according to the following format:

```

namespace new_name = current_name;

```

Namespace std

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.

Unnamed Namespaces

An unnamed-namespace-definition behaves as if it were replaced by:

```
namespace unique { namespace-body }  
using namespace unique;
```

Each unnamed namespace has an identifier, represented by unique, that differs from all other identifiers in the entire program. For example:

```
namespace { int i; }           // unique::i  
void f() { i++; }              // unique::i++  
  
namespace A {  
    namespace {  
        int i;           // A::unique::i  
        int j;           // A::unique::j  
    }  
}  
  
using namespace A;  
  
void h()  
{  
    i++;           //error: unique::i or A::unique::i  
    A::i++;        // error: A::i undefined  
    j++;           // A::unique::j++  
}
```

Unnamed namespaces are a superior replacement for the static declaration of variables. They allow variables and functions to be visible within an entire translation unit, yet not visible externally. Although entities in an unnamed namespace might have external linkage, they are effectively qualified by a name unique to their translation unit and therefore can never be seen from any other translation unit.