- Namespace defined:
  A collection of name definitions
  - Class definitions
  - Variable declarations
- Programs use many classes, functions
  - Commonly have same names
  - Namespaces deal with this
  - Can be "on" or "off"
    - If names might conflict ◊ turn off

using Directive

- using namespace std;
  - Makes all definitions in std namespace available
- Why might you NOT want this?
  - Can make cout, cin have non-standard meaning
    - Perhaps a need to redefine cout, cin
  - Can redefine any others

- namespace std contains all names defined in many standard library files
- Example:
  #include <iostream>
  - Places all name definitions (cin, cout, etc.) into std namespace
  - Program doesn't know names
  - Must specify this namespace for program to access names

Global Namespace

- All code goes in some namespace
- Unless specified -> global namespace
  - No need for using directive
  - Global namespace always available
  - Implied "automatic" using directive

Multiple Names

- What if name defined in both?
  - Error
  - Can still use both namespaces
  - Must specify which namespace used at what time

Specifying Namespaces

- Given namespaces NS1, NS2
  - Both have void function myFunction() defined differently

```
{
    using namespace NS1;
    myFunction();
}

{
    using namespace NS2;
    myFunction();
}
```

Creating a Namespace

- Use namespace grouping:
  namespace Name_Space_Name
  {
  Some_Code
  }
- Places all names defined in Some_Code into namespace Name_Space_Name
- Can then be made available:
  using namespace Name_Space_Name

See example

- Function declaration:

```
namespace Space1
{
    void greeting();
}
```

- Function definition:

```
namespace Space1
{
    void greeting()
    {
        cout << "Hello from namespace Space1.\n";
    }
}
```

- **Consider:**
  Namespaces NS1, NS2 exist
  Each have functions fun1(), fun(2)
  - **Declaration syntax:**
    using Name_Space::One_Name;
  - **Specify which name from each:**
    using NS1::fun1;
    using NS2::fun2;

INTRODUCTION TO NAMESPACES - PART 1

Namespaces are a relatively new C++ feature just now starting to appear in C++ compilers. We will be describing some aspects of namespaces in subsequent newsletters.

What problem do namespaces solve? Well, suppose that you buy two different general-purpose class libraries from two different vendors, and each library has some features that you'd like to use. You include the headers for each class library:

```
#include "vendor1.h"
#include "vendor2.h"
```

and then it turns out that the headers have this in them:

```
// vendor1.h

... various stuff ...

class String {
        ...
};

// vendor2.h

... various stuff ...

class String {
        ...
};
```

This usage will trigger a compiler error, because class String is defined twice. In other words, each vendor has included a String class in the class library, leading to a compile-time clash. Even if you could somehow get around this compile-time problem, there is the further problem of link-time clashes, where two libraries contain some identically-named symbols.

The namespace feature gets around this difficulty by means of separate named namespaces:

```
// vendor1.h

... various stuff ...

namespace Vendor1 {
```

```
        class String {
            ...
        };
}

// vendor2.h

... various stuff ...

namespace Vendor2 {
        class String {
            ...
        };
}
```

There are no longer two classes named String, but instead there are now classes named Vendor1::String and Vendor2::String. In future discussions we will see how namespaces can be used in applications.

INTRODUCTION TO NAMESPACES - PART 2

In the last issue we discussed one of the problems solved by namespaces, that of conflicting class names. You might be using class libraries from two different vendors, and they both have a String class in them. Using namespaces can help to solve this problem:

```
namespace Vendor1 {
        class String { ... };
}

namespace Vendor2 {
        class String { ... };
}
```

How would you actually use the String classes in these namespaces? There are a couple of common ways of doing so. The first is simply to qualify the class name with the namespace name:

```
Vendor1::String s1, s2, s3;
```

This usage declares three strings, each of type Vendor1::String.
Another approach is to use a using directive:

```
using namespace Vendor1;
```

Such a directive specifies that the names in the namespace can be used in the scope where the using directive occurs. So, for example, one could say:

```
using namespace Vendor1;


String s1, s2, s3;
```

and pick up the String class found in the Vendor1 namespace.
What happens if you say:

```
using namespace Vendor1;
using namespace Vendor2;
```

and both namespaces contain a String class? Will there be a conflict? The answer is "no", unless you try to actually use String. The using directive doesn't add any names to the scope, but simply makes them available. So usage like:

```
String s1;
```

would give an error, while:

```
Vendor1::String s1;
Vendor2::String s2;
```

would still work.
You might have noticed that namespaces have some similarities of notation with nested classes. But namespaces represent a more general way of grouping types and functions. For example, if you have:

```
class A {
    void f1();
    void f2();
};
```

then f1() and f2() are member functions of class A, and they operate on objects of class A (via the "this" pointer). In contrast, saying:

```
namespace A {
    void f1();
    void f2();
}
```

is a way of grouping functions f1() and f2(), but no objects or class types are involved.

INTRODUCTION TO NAMESPACES - PART 3

In previous issues we talked about how C++ namespaces can be used to group names together. For example:

```
namespace A {
    void f1();
```

```
        void f2();
    }

    namespace B {
        void f1();
        void f2();
    }
```

The members of the namespace can be accessed by using qualified names, for example:

```
    void g() {A::f1();}
```

or by saying:

```
    using namespace A;
    void g() {f1();}
```

Another interesting aspect of namespaces is that of the unnamed namespace:

```
    namespace {
        void f1();
        int x;
    }
```

This is equivalent to:

```
    namespace unique_generated_name {
        void f1();
        int x;
    }
    using namespace unique_generated_name;
```

All unnamed namespaces in a single scope share the same unique name. All global unnamed namespaces in a translation unit are part of the same namespace and are different from similar unnamed namespaces in other translation units. So, for example:

```
    namespace {
        int x1;
        namespace {
            int y1;
        }
    }

    namespace {
        int x2;
        namespace {
```

```
        int y2;
    }
}
```

x1 and x2 are in the same namespace, as are y1 and y2.
Why is this feature useful? It provides an alternative to the keyword "static" for controlling global visibility. "static" has several meanings in C and C++ and can be confusing. If we have:

```
static int x;
static void f() {}
```

we can replace these lines with:

```
namespace {
    int x;
    void f() {}
}
```

INTRODUCTION TO C++ NAMESPACES - PART 4

Previously we talked about how namespaces can be used to group names into logical divisions:

```
namespace Vendor1 {
    class String {
        ...
    };
    int x;
}

namespace Vendor2 {
    class String {
        ...
    };
    int x;
}
```

and how those names can be accessed via qualification:

```
Vendor2::String s;
```

or a using directive:

```
using namespace Vendor2;
```

```
String s;
```

Another way of accessing names is to employ a using declaration:

```
using Vendor2::String;

String s;
```

This can be a little confusing. A using directive:

```
using namespace X;
```

says that all the names in namespace X are available for use, but none of them are actually declared or introduced. A using declaration, on the other hand, actually introduces a name into the current scope. So saying:

```
using namespace Vendor2;
```

makes String and x available for use, but doesn't declare them. Saying:

```
using Vendor2::String;
```

actually introduces Vendor2::String into the current scope as a declaration. Saying:

```
using Vendor1::String;
using Vendor2::String;
```

will trigger a "duplicate declaration" compiler error.
There are several other aspects of using declarations that are worth learning about; these can be found in a good C++ reference book.