Namespaces

> *"The road to hell is paved with global variables"* - Steve McConnell

The purpose of this handout is give you an introductory, "big picture" presentation on namespaces. After reading this document, you should be able to:

1. Understand the purpose of namespaces in C++. In other words, what problem from C was solved by inventing them?
2. Have knowledge of when namespaces aren't very useful.
3. Understand advantages of unnamed or anonymous namespaces.
4. Explain `using` declarations and `using` directives.
5. Know the meaning of different types of headers such as `<iostream.h>`, `<iostream>`, `<string>`, `<string.h>`, and `<cstring>` in the C++ standard library namespace `std`.

It is important to understand the concept of namespaces because they show up everywhere. Although namespaces are not a sub-topic of generic programming, they become important in the presence of function templates. For example, making sense of a topic called argument-dependent lookup relies on a good understanding of namespaces.

References

1. Section 18.2 of the required text C++ Primer explains namespaces.
2. Microsoft provides a short introduction to namespaces while this page provides coverage related to syntax.

What is a variable? What is an object?

Let's begin by understanding what a variable is and what an object is. A *variable* is an *object* that has a *name*. An object, in turn, is a part of the computer's memory that has a *type*. This distinction is important because it is possible to have objects that don't have names. This happens when intermediate objects are created when expressions are evaluated or when objects have dynamic storage duration. An example with built-in and user-defined types:

```
int add(int lhs, int rhs) {
  return lhs + rhs;
}

std::string add(std::string lhs, std::string rhs) {
  return lhs + rhs;
}

int main() {
  std::cout << add(int {2}, int {5}) << "\n";
  std::cout << add(std::string{"Hello"}, std::string{"World"}) << "\n";
}
```

In this example, the arguments in the first call to function `add` are evaluated to unnamed objects of type `int` which are then used to initialize parameters `lhs` and `rhs` of function `add`. The expression `lhs+rhs` further evaluates to an unnamed object of type `int` which is returned by value. Thus, the call to function `add` will evaluate to an unnamed object of type `int`. The only variables encountered in this code are the two parameters of function `add`.

The second call to function `add` is also evaluated in the same manner except that `std::strig` is the type of objects and variables.

Scope and linkage

The *scope* or *visibility* of a variable is the region of program text over which the name is visible and can be referenced by other entities. Variables defined outside a function are called *external variables* and the scope of an external variable lasts from the point at which it is declared in the file to the end of the file being compiled. External variables are said to have *global scope*, and sometimes also referred to as *file scope*.

Linkage describes the accessibility of a variable between different source files or even within the same source file. There are three types of linkage: *no linkage*, *external linkage*, and *internal linkage*.

All variables declared inside a function including its arguments are *internal* or private to the function and can only be accessed from inside the function. Since these variables can never be accessed from other functions, we say that internal variables of a function have *no linkage*.

All external variables have the default property that all references to them by the same name, from *any* source file - even from a source file different than the one in which the external variable is defined - are references to the same thing. This property is called *external linkage*. External linkage implies that a program consisting of many source files must have *one and only one definition of an external variable* in these source files. However, there could be as many compatible declarations as required of the external variable. In C++, this rule is called the *One Definition Rule* (or, ODR).

The ability of allowing an external variable in a source file be accessible to other functions in the same source file but inaccessible to functions in other source files is called *internal linkage*. The author of a source file can ensure internal linkage for an external variable by adding keyword `static` to the declaration specifier.

Problems with global scope

The biggest issue with the global scope is that there's only one of them. In small programs, this is not a problem, since one programmer may produce all of these names. When *programming in the large*, there is usually a bevy of people putting names in this singular scope, and invariably this leads to name conflicts. For example, `library1.h` sourced from one library vendor might define a number of constants, including the following:

```
double const LIB_VERSION = 1.204;
```

Ditto for `library2.h` that is sourced from a different library vendor:

```
int const LIB_VERSION = 3;
```

It doesn't take great insight to see that there is going to be a problem if one of your source file tries to include both `library1.h` and `library2.h`. Unfortunately, outside of cursing under your breath, sending hate mail to the library authors, and editing the header files until the name conflicts are eliminated (which may not be possible with libraries since you don't have access to source files), there is little you can do about this kind of problem.

A similar situation would arise with variable and function names declared at global scope in different source files of the same program. Consider the following source file `mine.cpp` that defines four names in global scope:

```
#include <iostream>   // std::cout

int counter  {1};     // counter in global scope
int strength {2};     // strength in global scope

int Div2(int value) { // Div2 in global scope
  return value / 2;
}

int main() { // main in global scope
  std::cout << counter << "\n";  // use global counter
  std::cout << strength << "\n"; // use global strength
  std::cout << Div2(8) << "\n";  // use global Div2
}
```

Consider another source file `yours.cpp` that also defines a global variable named `strength`:

```
double strength {22.33};         // strength in global scope

double incr_strength(double x) { // incr_strength in global scope
  return x + strength;
}

// other stuff here ...
```

Both source files independently compile without flagging any errors. However, in the process of creating an executable program, the linker will flag an error:

```
/usr/bin/ld: yours.o:(.data+0x0): multiple definition of `strength'; mine.o:
(.data+0x0): first defined here
collect2: error: ld returned 1 exit status
```

The particular issue faced by the linker is that it is unable to reconcile the multiple definitions of external variable `strength` - one definition in `mine.cpp` and the second definition in `yours.cpp`. Recall the ODR rule requires a program have *one and only one definition of an external variable*.

Programmers try to reduce the possibility of name clashes by prepending some hopefully-unique prefix to each of their global names. Rather than risk having multiple definitions of name `strength` in the global scope, the author of `mine.cpp` might name the external variable as `hlpr1_strength` while another author might name their external variable as `hlpr2_strength`. Surely you must admit that the resulting names are less than pleasing to use and hard to understand.

It is possible that certain external variables only require internal linkage rather than external linkage. Programmers can reduce the possibility of name clashes by making external variables in source files have internal linkage by adding keyword `static` to the declaration statements of these variables. After amending `yours.cpp` so that external variable `strength` has internal linkage, it is now possible to satisfy the ODR rule and remove the previous linker error.

```
// revised version of yours.cpp
static double strength {22.33}; // make name strength private to this file

double incr_strength(double x) {
  return x + strength;
}
```

Namespaces

To avoid name clashes in the global scope, we have seen two C-style options:

- Prepend some hopefully-unique prefix to each global name
- Make certain global names private to the source file by adding keyword `static` to the declaration specifier of the variable. Such an external variable will now have internal linkage.

The first option makes names less than pleasing to use and hard to understand while providing the required external linkage for the variables and functions defined with these names. The second option enforces internal linkage of variables and function to a certain source file preventing their use by other source files. Is there a better solution that avoids name clashes for global variables in a program while allowing both uncluttered names and external linkage? C++ *namespaces* avoid name clashes and yet provide programmers the best of both worlds: define global variables having both external linkage and uncluttered, pleasing, and easy to understand names.

Namespace definitions

At its core, a `namespace` is just a fancy way of introducing a new scope that lets you use the simplest and most easy-to-understand names without confusing other programmers. Further, namespaces can be used to control whether a name has internal or external linkage. The general form of a namespace definition is:

```
namespace user-defined-name { // user-defined-name introduces a new scope
  declaration/definition
  declaration/definition
  ...
}
```

- A namespace can only be defined at the global scope or within another namespace. This means you can't define local namespaces inside functions nor `struct`s nor `class`es).
- `user-defined-name` must be *unique* in the global namespace. If `user-defined-name` is the name of a previously defined namespace, then you're adding declarations or definitions to the pre-existing namespace. This also means that namespaces can be *discontiguous*.
- Any declaration or definition that can appear in the global scope can appear in a user-defined namespace. This includes `class`es, `struct`s, declaration and definitions of functions and variables, templates, and other namespaces (nested).

Accessing names in namespaces

Suppose, the code in `mine.cpp` handles the implementation of mathematical types and functions for a software project. The names in the global scope can then be put in a unique namespace called `helpers`:

```
namespace helpers { // namespace helpers creates new scope

int counter  {1};
int strength {2};
int Div2(int value) {
  return value / 2;
}

} // like code blocks, namespaces don't end with a semicolon!!!
```

However, placing these names in a namespace will prevent `mine.cpp` from compiling:

```
namespace helpers { // namespace helpers creates new scope

int counter  {1};
int strength {2};
int Div2(int value) {
  return value / 2;
}

} // like code blocks, namespaces don't end with a semicolon!!!

int main() { // main in global scope
  std::cout << counter << "\n";  // error!!! name counter is not declared
  std::cout << strength << "\n"; // error!!! name strength is not declared
  std::cout << Div2(8) << "\n";  // error!!! name Div2 is not declared
}
```

Clients can access names in namespace `helpers` in any of three ways: by importing all names in a namespace into a scope; by importing individual names into a scope; or by explicitly qualifying a name for one-time use.

Here is an example of importing all names in namespace into a scope:

```
int main() { // main in global scope
  using namespace helpers; // make all names in helpers available without
                           // qualification in this scope
  std::cout << counter << "\n";
  std::cout << strength << "\n";
  std::cout << Div2(8) << "\n";
}
```

Here is an example of importing individual names in namespace into a scope. Since namespace `helpers` introduces a unique scope, we need to *qualify* the names in the namespace:

```
int main() { // main in global scope
  using helpers::counter;        // make only name counter available without
  std::cout << counter << "\n";  // qualification in this scope
  using helpers::strength;       // make only name strength available without
  std::cout << strength << "\n"; // qualification in this scope
  using helpers::Div2;           // make only name Div2 available without
  std::cout << Div2(8) << "\n";  // qualification in this scope
}
```

Here is an example of explicitly qualifying a name for one-time use. The name `Div2` is visible only in scope `helpers` and to reference the name in global scope, the name must be qualified with the namespace:

```
int main() { // main in global scope
  std::cout << helpers::counter << "\n";  // for one-time use
  std::cout << helpers::strength << "\n"; // ditto
  std::cout << helpers::Div2(8) << "\n";  // ditto
}
```

Namespace definitions don't have to be contiguous

Namespace definitions do not have to be contiguous:

```
namespace helpers {
int counter  {1};
int strength {2};
}

// other code here ...

namespace helpers {
int Div2(int value) {
 return value / 2;
}
}
```

However, all namespace members must be declared before their first use:

```
1  namespace helpers {
2  int counter  {1};
3  int strength {2};
4  }
5
6  int main() { // main in global scope
7    std::cout << helpers::counter << "\n";  // ok: helpers::counter declared
8    std::cout << helpers::strength << "\n"; // ok: helpers::strength declared
9    std::cout << helpers::Div2(8) << "\n";  // error: math::Div2 not declared
10 }
11
12 namespace helpers {
13 int Div2(int value) { // definition
14  return value / 2;
15 }
16 }
```

Line 9 will not compile since `math::Div2` is declared *after its first use*.

Interfaces and implementations

We declare names in one namespace definition and define them in another. This is now OK in `mine.cpp`:

```
namespace helpers {
```

```
extern int counter;  // declaration
extern int strength; // ditto
int Div2(int);       // ditto
}

int main() { // main in global scope
  std::cout << helpers::counter << "\n";
  std::cout << helpers::strength << "\n";
  std::cout << helpers::Div2(8) << "\n";  // ok, compiles and links
}

namespace helpers {
int counter  {1}; // definition
int strength {2}; // ditto
int Div2(int value) { // definition
 return value / 2;
}
}
```

Note also that the separate definitions of the same namespace (as above) can be in separate files as well. They don't have to be in the same physical source file (and often they won't be). This gives you the flexibility to put the interface for your code into the public files (header files) where your users can see it, and keep the implementation hidden in source files.

A header file `helpers.h` containing the interface looks like this:

```
namespace helpers {
extern int counter;  // need declaration not definition
extern int strength; // ditto
int Div2(int);       // ditto
}
```

The implementation would look like this in source file `helpers.cpp`:

```
namespace helpers {
int counter  {1}; // definition
int strength {2}; // ditto

int Div2(int value) { // ditto
 return value / 2;
}
}
```

You can now use the interface in `mine.cpp` like this:

```
#include <iostream>
#include "helpers.h"

int main() { // main in global scope
  std::cout << helpers::counter << "\n";  // ok, compiles
  std::cout << helpers::strength << "\n"; // ditto
  std::cout << helpers::Div2(8) << "\n";  // ditto
}
```

Accessing same name from more than one namespace

Now lets turn our attention to `yours.cpp`. Recall the author of `yours.cpp` used keyword `static` to make external variable `strength` have internal linkage:

```
// revised version of yours.cpp
static double strength {22.33}; // make name strength private to this file

double incr_strength(double x) {
  return x + strength;
}
```

What happens if the author deletes the keyword `static` and makes external variable `strength` have external linkage? The program consisting of source files `mine.cpp`, `helpers.cpp`, and `yours.cpp` will compile and link because names `strength` and `helpers::strength` exist in global scope and namespace scope `helpers`, respectively. That is, a new scope `helpers` has been carved out of the global scope. To avoid future name clashes (as more source files are added to the project), the external name `strength` in `yours.cpp` is declared in a unique namespace, say `phys`:

```
namespace phys { // namespace phys creates new scope

double strength {22.33}; // make name strength private to this file
double incr_strength(double x) {
  return x + strength;
}

}
```

Using the idea of interfaces and implementations, let's provide the declarations for names in namespace `phys` in header file `yours.h`:

```
namespace phys { // namespace phys creates new scope

extern double strength;        // need declaration not definition
double incr_strength(double); // ditto
}
```

and provide the definitions in source file `yours.cpp`:

```
#include "yours.h"

namespace phys {
double strength {22.33};          // definition
double incr_strength(double x) { // ditto
  return x + strength;
}
}
```

One of the nicest things about namespaces is that you can use the `strength` variables defined in namespaces `helpers` and `phys` without conflict, provided you explicitly say which namespace's `strength` you wanted:

```
// mine.cpp
#include <iostream>
#include "helpers.h"
#include "yours.h"

int main() { // main in global scope
  std::cout << helpers::counter << "\n";  // ok
  std::cout << strength << "\n";          // error!!! which strength?
  std::cout << helpers::strength << "\n"; // fine, no ambiguity
  std::cout << phys::strength << "\n";    // also no ambiguity
  std::cout << helpers::Div2(8) << "\n";  // ok
}
```

Scope resolution operator `::`

Consider the following code fragment:

```
1 #include <iostream>
2
3 int main() {
4   std::cout << "Hello World!!!\n";
5 }
```

The name `std::cout` on line 4 is a *qualified name*, which uses the [scope resolution operator](#) `::`. To the left of the `::` is the (possibly qualified) name of a scope which in the case of `std::cout` is the namespace `std`. To the right of the `::` is a name that is defined in the scope named on the left. Thus, `std::cout` means the "name `cout` that is in the namespace scope `std`."

As shown in the following code fragment, the `::` operator allows you to access hidden global variables:

```
#include <iostream> // std::cout

int foo {1}; // foo in global scope
int bar {2}; // ditto

void func() {
  int foo {10};    // local foo #1 hides global foo
  int bar {foo};   // local bar #1 hides global bar
  int baz {::foo}; // local baz #1 initialized with global foo

  if (bar == 10) {  // local bar #1
    int foo {100};  // local foo #2 hides local #1 and global
    bar = foo;      // local bar #1 is set to local foo #2
    foo = ::bar;    // local foo #2 is set to global bar
  }

  ::foo = foo;   // global foo assigned value of local foo #1
  ::bar = ::foo; // global bar assigned value of global foo

  std::cout << "foo is " << foo << "\n";     // local foo #1 is 10
  std::cout << "bar is " << bar << "\n";     // local bar #1 is 100
  std::cout << "::foo is " << ::foo << "\n"; // global foo is 10
  std::cout << "::bar is " << ::bar << "\n"; // global bar is 10
}
```

The definition of variable `foo` in line 7 *hides* global variable `foo`; that is, any references to name `foo` in the scope of function `func` are to the nearest valid declaration of name `foo` (which is the declaration on line 7). In C, if a global variable is hidden by a new declaration with the same name, there is no way to access the global variable. In C++, the `::` operator makes it possible to access a hidden global variable. If the left of the `::` is empty then the right is a name that is defined in the global scope of the program. Thus the initializing expression on line 9 `::foo` means the "name `foo` that is in the program's global scope."

The definition of `foo` on line 12 hides variable `foo` declared on line 7 (which in turn hides global variable `foo` declared on line 3). Any references to name `foo` in the `if` statement block are to the `foo` declared on line 12. Again, you can access the global variable `foo` using expression `::foo`. However, there is no way to access the hidden variable on line 7. This means that if you hide a name in an outer scope, you can never refer to it unless the hidden name was global. That is, there is no way to access any hidden names in any intermediate scope.

Nested namespaces

Eventually, as your programs get larger and larger, even a namespace is going to have hundreds or thousands of names. Having everything in a single namespace may lead to conflicts. Names associated with specific portion of the program (such as graphics library or numerical library or I/O library) can be put into related namespaces. Just like we partitioned the global scope using namespaces, we can partition a namespace's scope by nesting these related namespaces within the outer namespace to create a hierarchy of namespaces. This idea of nested namespaces is somewhat similar to nested structures. Here is an example:

```cpp
#include <iostream> // std::cout

namespace DigiPen {
  int Div2(int x) {return x / 2;}

  namespace IntroProg {
    int Div2(int x) {return x / 2;}
  }

  namespace AdvProg {
    int Div2(int x) {return x >> 1;}
  }
}

int main() {
  std::cout << DigiPen::Div2(8) << "\n";
  std::cout << DigiPen::IntroProg::Div2(8) << "\n";
  std::cout << DigiPen::AdvProg::Div2(8) << "\n";
}
```

Unnamed namespaces

What happens when we run out of unique names for namespaces? It's unlikely to happen, but as more and more code uses namespaces, the chances for a collision are high. Namespace names are global, so there's no way to protect them from other global names. This is a problem if code uses lots of small namespaces. We could come up with some kind of globally unique ID scheme to guarantee unique namespaces:

```
namespace NS_1E266980_A661_48B6_94D1_C9DEA80A328B {
  // stuff
}

namespace NS_6FB60AE7_AEEE_4285_88A7_6F0C28B34B5B {
  // other stuff
}
```

This option makes names that are cluttered, less than pleasing to use, and hard to understand. A better approach is *unnamed namespaces*:

```
1  #include <iostream> // std::cout
2  #include <cmath>    // std::sqrt
3
4  namespace {
5    double my_sqrt(double x) { return std::sqrt(x); }
6  }
7
8  int main() {
9    std::cout << my_sqrt(25.0) << "\n";   // no qualification needed
10   std::cout << ::my_sqrt(25.0) << "\n"; // my_sqrt in global scope
11 }
```

The `my_sqrt` function is defined in an unnamed namespace. As shown on line 9, no qualification is required to refer to function `my_sqrt`. Likewise, as shown on line 10, the scope operator `::` can also be used to refer to function `my_sqrt`.

If we have a name in an unnamed namespace that is the same as a global name in our program, we won't be able to access the name in the unnamed namespace.

```
1  #include <iostream> // std::cout
2  #include <cmath>    // std::sqrt
3
4  namespace {
5    double my_sqrt(double x) { return std::sqrt(x); }
6  }
7
8  double my_sqrt(double x) { return std::sqrt(x); } // global
9
10 int main() {
11   // error: is it from unnamed namespace or line 8?
12   std::cout << my_sqrt(25.0) << "\n";
13   // calling global function on line 8
14   std::cout << ::my_sqrt(25.0) << "\n"; // ok
15 }
```

Names in an unnamed namespace are local to the file in which the unnamed namespace is defined. By having similar names in anonymous namespaces in different source files, we can keep these names private to each source file. This is similar to using keyword `static` to provide internal linkage to global variables and functions in a source file.

> ***Design tip: To provide internal linkage to external variables and functions, prefer to use unnamed namespaces over the already-overused keyword*** `static`.

Namespace aliases

Given these namespaces:

```
namespace AdvancedProgramming {
  int foo {11};
  int bar {12};
  int f1(int x)  { return x / 2; }
}

namespace IntroductoryProgramming {
  int foo {21};
  int bar {22};
  int Div2(int x) {return x / 2; }
}
```

using them requires a lot of typing:

```
int main() {
  std::cout << AdvancedProgramming::foo << "\n";
  std::cout << IntroductoryProgramming::Div2(8) << "\n";
}
```

To allow unique namespaces and to shorten the names, you can create a *namespace alias*:

```
1  // declare these after the namespace definitions above
2  namespace AP = AdvancedProgramming;
3  namespace IP = IntroductoryProgramming;
4
5  int main() {
6    // now, use the shorter aliases
7    std::cout << AP::foo << "\n";
8    std::cout << IP::foo << "\n";
9    std::cout << AP::f1(8) << "\n";
10   std::cout << IP::Div2(8) << "\n";
11 }
12
13 void func() {
14   // you can "re-alias" a namespace (must be in different scope)
15   namespace AP = IntroductoryProgramming;
16
17   std::cout << AP::f1(8) << "\n";   // now, an error no f1 in AP
18   std::cout << AP::Div2(8) << "\n"; // ok
19   std::cout << IP::Div2(8) << "\n"; // same as above
20 }
```

- Aliases defined outside of a function are still only visible within the file they were defined (i.e. they are not global).
- You can "re-alias" a namespace later in the code, as shown on line 15. However, the redefinition must happen in a new scope. Otherwise, it is an illegal redefinition.
- Once you create an alias, you can't "un-create" it. You can only re-alias it.

> ***Programming tip: Overuse of namespace aliases may also lead to confusion.***

> ***Programming tip: Be careful when you redefine an alias as it may just lead to more confusion.***

Aliases for nested namespaces

Creating aliases for nested namespaces as well:

```
1  namespace AdvancedProgramming {
2    int Div2(int x) {return x >> 1;}
3  }
4
5  namespace AP = AdvancedProgramming; // alias for global namespace
6
7  namespace DigiPenInstituteOfTechnology {
8    int Div2(int x) { return x / 2; }
9
10   namespace IntroductoryProgramming {
11     int Div2(int x) { return x / 2; }
12   }
13
14   namespace AdvancedProgramming {
15     int Div2(int x) { return x >> 1; }
16   }
17 }
18
19 namespace DIT    = DigiPenInstituteOfTechnology;
20 namespace DIT_IP = DigiPenInstituteOfTechnology::IntroductoryProgramming;
21 namespace DIT_AP = DIT::AdvancedProgramming;  // uses previous alias
22
23 // possible to have multiple aliases
24 namespace CS120 = DIT::IntroductoryProgramming;
25 namespace CS225 = DIT::AdvancedProgramming;
26
27 int main() {
28   // these are all equivalent
29   std::cout <<
30     DigiPenInstituteOfTechnology::IntroductoryProgramming::Div2(8) << "\n";
31   std::cout << DIT::IntroductoryProgramming::Div2(8) << "\n";
32   std::cout << DIT_IP::Div2(8) << "\n";
33   std::cout << CS120::Div2(8) << "\n";
34
35   // these are equivalent
36   std::cout << DIT_AP::Div2(8) << "\n";
37   std::cout << CS225::Div2(8) << "\n";
38 }
```

Note that you can't do this:

```
std::cout << DIT::AP::Div2(8) << "\n";
```

because `AP` is not a member of `DIT` namespace. That is, the alias on line 5 cannot be used like a preprocessor #define`.

Another example:

```
namespace DigiPenInstituteOfTechnology {
```

```cpp
  namespace GAM400 {
    namespace Graphics {
      void initialize();
      // other stuff here
    }

    namespace Physics {
      void initialize();
      // other stuff here
    }

    namespace Network {
      void initialize();
      // other stuff here
    }
  }
}
```

Now, with one alias like this:

```cpp
namespace DIT = DigiPenInstituteOfTechnology;
```

You can access symbols inside the hierarchy something like this:

```cpp
DIT::GAM400::Graphics::initialize();
DIT::GAM400::Physics::initialize();
DIT::GAM400::Network::initialize();
```

> ***Design tip: Don't create very terse namespaces like `std`. Create unique and meaningful namespaces and let the user create shorthand notation with aliases.***

`using` declarations

Like any other declaration, an `using` *declaration* introduces names declared in a namespace scope in the current scope. It allows you to make specific names declared in a namespace accessible *without* requiring the namespace and scope resolution operator. Example:

```cpp
namespace Stuff {
  int foo {11}; // Stuff::foo
  int bar {12}; // Stuff::bar
  int baz {13}; // Stuff::baz
}

namespace Stuff2 {
  int bar {111}; // Stuff2::bar
}

// name foo accessible in rest of file scope without namespace qualifier
using Stuff::foo;

void f1() {
  Stuff::foo = 21;   // ok: using namespace
  foo = 22;          // ok: namespace not required
  using Stuff::bar;  // make bar available in this scope only
  bar = 30;          // ok
```

```
19   int bar {5};       // error: redeclaring name bar declared in line 17
20   using Stuff2::bar; // error: redeclaring name bar declared in line 17
21 }
22
23 void f2() {
24   int foo {3};     // This is a new foo, it hides Stuff::foo
25   Stuff::foo = 4; // ok, using qualified name
26   ::foo = 5;       // ok, global foo (i.e. Stuff::foo)
27 }
28
29 int foo {222}; // error: redeclaring name foo declared in line 12
30
31 int main() {
32   foo = 23;          // ok because of using declaration on line 12
33   bar = 30;          // error: name bar not declared in this scope
34   using Stuff::baz; // make baz available in this function only
35   baz = 40;          // ok
36 }
```

Some salient points about the above code:

- `using` declarations *declare* a name, which means you can't redeclare/redefine with the same name *in the same scope*, as shown below on lines 19, 20, and 29.
- If many functions in the file need access to a name in a namespace, you should probably put the `using` declaration at the top of the file, outside of any function, as is done on line 12.
- An `using` declaration placed outside of a function has *file scope* not global scope.
- If only a few functions need the name, you should put the `using` declaration in the function(s) where the name is needed, as is done on lines 17 and 34.

`using` directives

An *using directive* allows you to make all names in a namespace visible at once. Assume we have these names in a namespace:

```
namespace Stuff {
  int foo {11};  // Stuff::foo
  int bar {12};  // Stuff::bar
  int baz {13};  // Stuff::baz
}
```

We can make them all accessible with a `using` directive:

```
// every name in Stuff is visible from here down in the file
using namespace Stuff;

int main() {
  std::cout << foo << "\n";  // Stuff::foo
  std::cout << bar << "\n";  // Stuff::bar
  std::cout << baz << "\n";  // Stuff::baz
}
```

`using` directives are scoped; they apply only within the block where the directive is specified:

```cpp
int main() {
  // every name in Stuff is visible only in this scoe
  using namespace Stuff;

  std::cout << foo << "\n";  // Stuff::foo
  std::cout << bar << "\n";  // Stuff::bar
  std::cout << baz << "\n";  // Stuff::baz
}

// unqualified members in Stuff not available here.
```

Ambiguity errors are detected when an ambiguous name is referenced, not when the directive is encountered:

```cpp
namespace Stuff2 {
  int bar {111}; // Stuff2::bar
}

using namespace Stuff;  // make all names from Stuff accessible
using namespace Stuff2; // make all names from Stuff2 accessible

foo = 10; // ok, Stuff::foo
bar = 30; // error: which bar? this is ambiguous.
```

Of course, qualified names can override the `using` directive.

```cpp
Stuff::bar = 100;  // OK
Stuff2::bar = 200; // OK
```

More detailed example:

```cpp
namespace Stuff {
  int foo {11};       // Stuff::foo
  int bar {12};       // Stuff::bar
  int baz {13};       // Stuff::baz
}

void f1() {
  int foo {3};        // local, hides nothing
  int x {Stuff::foo}; // ok
  int y {bar};        // error, bar is unknown
}

int foo {20};         // global ::foo

int main() {
  using namespace Stuff; // Stuff's members accessible without
                         // qualifier Stuff:: in this function
  std::cout << ::foo << "\n";      // no problem, global
  std::cout << Stuff::foo << "\n"; // no problem, Stuff::foo
  std::cout << foo << "\n";        // ambiguity error: ::foo or Stuff::foo?

  std::cout << bar << "\n";  // Stuff::bar
  std::cout << baz << "\n";  // Stuff::baz

```

```
  int foo {3};         // ok: hides Stuff::foo and ::foo
  int x {Stuff::foo}; // ok: use qualified name
  x = foo;             // ok: local foo above
  x = ::foo;           // ok: global foo
}
```

Summarizing the above code:

- `using` directives are not meant to be used to make it "easier" on the programmer (by saving keystrokes).
- Many `using` directives will cause global namespace to be polluted, which is the primary purpose of namespaces to begin with.
- It's best to avoid `using` directives because they've the potential of polluting a scope by inserting all the names from the namespace.

> ***Programming tip: Neve use `using` directives in header files that are meant to be used by others. Aren't all header files for others to use?***

`std` namespace

C++'s standard library is incredibly big: out of the 1850 pages in the specification, the base language only takes about 450 pages while the library takes about 1400 pages. Because the library is big, it contains a lot of functionality. The other important detail about the library is that almost everything in it is a template. Because the library has so much in it, there's a reasonable chance you may choose a class or function name that's the same as a name in the standard library. To shield you from the name conflicts that would result, virtually everything in the standard library since C++98 (the first C++ ISO standard was formalized in 1998) is nestled in namespace `std`. But that lead to a new problem. Before C++98, gazillions of lines of C++ code were written with a pseudo-standard library that relied on functionality declared in headers `<iostream.h>`, `<complex.h>`, `<limits.h>`, `<stdlib.h>`, etc. That pseudo-standard library wasn't designed to use namespaces nor templates. To allow existing code to continue to compile with C++98 compilers, the C++98 standard decided to retain the pseudo-standard library for a period of time and create new headers for the `std`-wrapped components. The algorithm for creating new headers was trivial: the `.h` on existing C++ headers was simply dropped. So `<iostream.h>` became `<iostream>` in C++98 and future standards. For C headers, the same algorithm was applied, but a `c` prepended to each result. Hence C's `<string.h>` became `<cstring>`, `<stdio.h>` became `<cstdio.h>`. The old C++ headers were officially deprecated while the C headers are being maintained for compatibility with ISO C but are expected to be deprecated in future versions.

Practically speaking, this is the C++ header situation:

- Old C++ header names like `<iostream.h>` were officially deprecated in C++98 and therefore were removed from the official standard. The contents of such headers are *not* in namespace `std`. Compiler vendors supported these non-standard headers for many years to keep alive pre-1998 code. Currently, most mainstream compilers do not support pre-C++98 style headers.
- C++98 header names like `<iostream>` contain the same basic functionality as the corresponding old headers, but the contents of the headers *are* in namespace `std`.
- Standard C headers like `<stdio.h>` continue to be supported [but as mentioned earlier these headers are expected to be deprecated in future versions]. The contents of such headers are *not* in `std`.

- New C++ headers for functionality in the C library have names like `<cstdio>`. They offer the same contents as the corresponding old C headers, but the contents *are* in `std`.

All this seems a little weird at first, but it's really not that hard to get used to. The biggest challenge is keeping all the string headers straight: `<string.h>` is the old C header for `char*`-based string manipulation functions, `<string>` is the `std`-wrapped C++ header for C++ string classes, and `<cstring>` is the `std`-wrapped version of the old C header. If you can master that (and I know you can), the rest of the library is easy.

Now that we've seen some of the details of how namespaces are created and used, let's look at how they can be applied with `std` namespace. This code should be easy to understand now:

```
#include <iostream>  // for std::cout and std::endl
using namespace std; // for access to *all* names inside std namespace

int main() {
  cout << "Hello" << endl;
}
```

The `using` directive in the above code is not recommended, but typically seen in high-school C++ courses and throughout the web. A better way is to use `using` declarations to control which names in namespace `std` to bring into the program:

```
#include <iostream> // for std::cout and std::endl
using std::cout;    // using declaration, global scope
using std::endl;    // using declaration, global scope

int main() {
  cout << "Hello" << endl;  // std::cout and std::endl
}
```

Rather than polluting the file scope with names `cout` and `endl`, an even better way to write the above `using` declarations is to limit their scope to function `main`:

```
#include <iostream> // for std::cout and std::endl

int main() {
  using std::cout;  // using declaration, local scope
  using std::endl;  // using declaration, local scope

  cout << "Hello" << endl;  // std::cout and std::endl
}
```

I believe the preferred way is to limit polluting even the function scope and instead to write code that uses the C++ standard library using qualified names:

```
#include <iostream>  // for std::cout and std::endl

int main() {
  std::cout << "Hello" << std::endl;
}
```

Now we can write code like this (to ensure job security):

```cpp
#include <iostream> // for std::cout and std::endl

int main() {
  int cout {16};     // cout is an int
  cout = cout << 3; // multiply cout by 8
  std::cout << cout << 5 << std::endl;   // std::cout is a stream, prints: 163
  std::cout << (cout << 5) << std::endl; // std::cout is a stream, prints: 128
}
```

Of course, we would never write code like the above! But, there are thousands of names in the C++ global namespace, so the chances that you collide with one is pretty good. You should learn to take control over when, where, and how names are introduced into your programs. Don't introduce names "accidentally." C++ gives the programmer complete control, but this power is often abused (or not fully understood) by beginning C++ programmers. Namespaces were added to help simplify management of large programs. If you're writing a trivial, throw-away program, it's probably not a big deal if you have using directives outside of your functions.

Here's a classic example of the problem with a `using` directive:

```cpp
1 #include <algorithm> // STL algorithms
2 using namespace std; // make the whole C++ universe available!
3
4 int count = 0;
5
6 int increment() {
7   return ++count; // error, identifier count is ambiguous
8 }
```

The `using` directive in line 2 will introduce name `count` declared in namespace `std` to file scope. In that same scope, the programmer is defining variable `count`. Thus, the reference to name `count` in line 7 is ambiguous. Hopefully, that should convince you that `using` directives should be used judiciously.