Lecture Material

- Namespaces

Namespaces

- There is only one global namespace
- In programs written by many persons, or using the libraries written by others, there is a risk of name collision

```
// library1.h
const double LIB_VERSION = 1.204;
```

```
// library2.h
const int LIB_VERSION = 3;
```

```
// main.c
#include <library1.h>
#include <library2.h>
```

- Traditionally, the problem is solved by using prefixes unique for each library

```
// library1.h
const double library1_LIB_VERSION
                          = 1.204;
```

```
// library2.h
const int library2_LIB_VERSION = 3;
```

Namespaces

- Generally, the namespaces replace the identifier prefixes
- Instead of

```
const double sdmBOOK_VERSION = 2.0;          // in this library,
class sdmHandle { ... };                     // each symbol begins
sdmHandle& sdmGetHandle();                   // with "sdm"
```

we write:

```
namespace sdm {
    const double BOOK_VERSION = 2.0;
    class Handle { ... };
    Handle& getHandle();
}
```

Using Namespaces

```
void f1() {
    using namespace sdm;        // make all symbols in sdm
                                // available w/o qualification
                                // in this scope
    cout << BOOK_VERSION;       // okay, resolves to sdm::BOOK_VERSION ...
    Handle h = getHandle();     // okay, Handle resolves to
                                // sdm::Handle, getHandle
    ...                         // resolves to sdm::getHandle
}
void f2() {
    using sdm::BOOK_VERSION;    // make only BOOK_VERSION
                                // available w/o qualification
                                // in this scope
    cout << BOOK_VERSION;       // okay, resolves to
                                // sdm::BOOK_VERSION
    ...
    Handle h = getHandle();     // error! neither Handle
                                // nor getHandle were
    ...                         // imported into this scope
}
void f3() {
    cout << sdm::BOOK_VERSION;// okay, makes BOOK_VERSION
                                // available for this one use
    ...                         // only
    double d = BOOK_VERSION;    // error! BOOK_VERSION is
                                // not in scope
    Handle h = getHandle();     // error! neither Handle
                                // nor getHandle were
    ...                         // imported into this scope
}
```

Using Namespaces

- In case of ambiguity, the scope resolution operator :: can be used

```
namespace AcmeWindowSystem {
     ...
     typedef int Handle;
     ...
}

void f() {
     using namespace sdm;                 // import sdm symbols
     using namespace AcmeWindowSystem; // import Acme symbols
     ...                                  // freely refer to sdm
                                          // and Acme symbols
                                          // other than Handle
     Handle h;                            // error! which Handle?
     sdm::Handle h1;                      // fine, no ambiguity
     AcmeWindowSystem::Handle h2;         // also no ambiguity
     ...
}
```

Anonymous Namespaces

- An anonymous namespace replaces the usage of keyword *static* before a global identifier

```
#include "header.h"

namespace {
   int a;
   void f() {/* ... */}
   int g() {/* ... */ }
}
```

- External access to the anonymous namespace is possible due to an implicit *using* directive. The above declaration is equivalent to the following:

```
#include "header.h"

namespace $$$ {
   int a;
   void f() {/* ... */}
   int g() {/* ... */ }
}

using namespace $$$;
```

- $$$ is some name which is unique in the scope, where this namespace is defined

Searching for Names

- A function with an argument of type T is frequently defined in the same namespace as T. If the function cannot be found in the context it is used, the namespaces of its arguments are searched.

```
namespace Chrono {
   class Date { /* ... */ };
   bool operator==(const Date&, const std: :string&) ;
   std::string format(const Date&) ; // make string representation
   // ...
}

void f(Chrono::Date d, int i)
{
   std::string s = format(d) ; // Chrono::format()
   std::string t = format(i) ; // error: no format() in scope
}
```

- This rule makes possible to avoid the explicit qualifying of names, what is particularly important in case of operator functions and templates, where is can be especially troublesome.

Searching for Names

- If a function takes arguments from more than one namespace, functions are searched for in the namespaces of every argument an the function overloading is resolved in an usual way.
- If a function is called by a member function, the member functions of the same class and its base classes have the priority over the functions found using the argument types.

```
namespace Chrono {
   class Date { /* ... */ };
   bool operator==(const Date&, const std::string&);
   std::string format(const Date&) ; // make string representation
   // ...
}

void f(Chrono::Date d, std::string s)
{
   if (d == s) {
       // ...
   }
   else if (d == "August 4, 1914") {
       // ...
   }
}
```

Namespace Aliases

- If the users are giving short names to the namespaces, they can cause the naming conflict

```
namespace A { // short name, will clash (eventually)
   // ...
}
A::String s1 = "Grieg";
A::String s2 = "Nielsen";
```

- Long names are inconvenient to use

```
namespace American_Telephone_and_Telegraph{ // too long
   // ...
}
American_Telephone_and_Telegraph::String s3 = "Grieg";
American_Telephone_and_Telegraph::String s4 = "Nielsen";
```

- This dilemma can be solved by using the short alias for the long namespace

```
// use namespace alias to shorten names:
namespace ATT = American_Telephone_and_Telegraph;
ATT::String s3 = "Grieg";
ATT::String s4 = "Nielsen";
```

Namespace Aliases

- Thanks to the namespace aliases, the user can also refer to the library functions.

```
namespace Lib = Foundation_library_v2r11;
// ...
Lib::set s;
Lib::String s5 = "Sibelius";
```

- It simplifies replacement of one library with another. Using *Lib* instead of *Foundation_library_v2r11* simpifies the transition to version *v3r5* - it is enough to change the initialization of *Lib* alias and recompile the code.
- Overuse of namespace aliases can lead to confusion.

Composition of Namespaces

- Frequently we want to create an interface based on existing interfaces, e.g.:

```
namespace His_string {
   class String { /* ... */ };
   String operator+(const String&, const String&) ;
   String operator+(const String&, const char*) ;
   void fill(char) ;
   // ...
}
namespace Her_vector {
   template<class T> class Vector { /* ... */ };
   // ...
}
namespace My_lib {
   using namespace His_string;
   using namespace Her_vector;
   void my_fct(String&) ;
}
```

- Now we can use *My_lib* during program development:

```
void f()
{
   My_lib::String s = "Byron"; // finds My_lib::His_string::String
   // ...
}
using namespace My_lib;
void g(Vector<String>& vs)
{
   // ...
   my_fct(vs[5]) ;
   // ...
}
```

Composition of Namespaces

- Defining the function or data, we need to know the real name of an entity:

```
void My_lib::fill() // error: no fill() declared in My_lib
{
   // ...
}
void His_string::fill() // ok: fill() declared in His_string
{
   // ...
}
void My_lib::my_fct(My_lib::Vector<My_lib::String>& v) // ok
{
   // ...
}
```

- The ideal namespace should:
  - express the logically coherent set of features
  - do not give the users access to unrelated features
  - do not burden the user with a complex notation

Selection

- Ocasionally, we want access to only a few names from a namespace. We could do that by writing a namespace declaration containing only names we want

```
namespace His_string{ // part of His_string only
  class String { /* ... */ };
  String operator+(const String&, const String&) ;
  String operator+(const String&, const char*) ;
}
```

- If we are not a designer of *His_string* namespace, this can easily get messy. A change in the "real" definition of *His_string* will not be reflected in this declaration. Selection of features from the namespace is more explicitly made with *using* declaration.

```
namespace My_string {
  using His_string::String;
  using His_string::operator+; // use any + from His_string
}
```

- A *using* declaration brings every declaration with a given name into scope. In particular, a single *using* declaration can bring in every variant of an overloaded function.

Composition and Selection

- Combining composition (by *using* directives) with selection (by *using* declarations) yields the flexibility needed for most realworld examples. With these mechanisms, we can provide access to a variety of facilities in such a way that we resolve name clashes and ambiguities arising from their composition.

```
namespace His_lib {
  class String { /* ... */ };
  template<class T> class Vector { /* ... */ };
  // ...
}
namespace Her_lib {
  template<class T> class Vector { /* ... */ };
  class String { /* ... */ };
  // ...
}
namespace My_ lib {
  using namespace His_lib;    // everything from His_lib
  using namespace Her_lib;    // everything from Her_lib
  using His_lib::String;      // resolve potential clash in favor of His_lib
  using Her_lib::Vector;      // resolve potential clash in favor of Her_lib
  template<class T> class List { /* ... */ };      // additional stuff
  // ...
}
```

Composition and Selection

- Names explicitly declared in a namespace (including names declared by *using* declarations) take priority over names made accessible in another scope by a *using* directive. The name in a new namespace can be changed using the *typedef* directive or inheritance.

```
namespace Lib2 {
  using namespace His_lib;    // everything from His_lib
  using namespace Her_lib;    // everything from Her_lib
  using His_lib::String;      // resolve potential clash in favor of His_lib
  using Her_lib::Vector;      // resolve potential clash in favor of Her_lib
  typedef Her_lib::String Her_string;      // rename
  template<class T> class His_vec          // "rename"
      : public His_lib::Vector<T> { /* ... */ };
  template<class T> class List { /* ... */ };      // additional stuff
  // ...
}
```

Namespaces and Overloading

- Overloading works across namespaces. This is essential to allow us to migrate existing libraries to use namespaces with minimal source code changes.

```
// old A.h:
void f(int) ;
// ...

// old B.h:
void f(char) ;
// ...

// old user.c:
#include "A.h"
#include "B.h"
void g()
{
  f('a') ; // calls the f() from B.h
}
```

```
// new A.h:
namespace A {
   void f(int) ;
   // ...
}
// new B.h:
namespace B {
   void f(char) ;
   // ...
}
// new user.c:
#include "A.h"
#include "B.h"
using namespace A;
using namespace B;
void g()
{
   f('a') ; // calls the f()from B.h
}
```

Namespaces Are Open

- A namespace is open; that is, you can add names to it from several namespace declarations. In this way, we can support large program fragments within a single namespace.

```
namespace A {
   int f() ; // now A has member f()
}
namespace A {
   int g() ; // now A has two members, f() and g()
}
```

- This openness is an aid in a program transition into using namespaces

```
// my header:
void f() ; // my function
// ...
#include<stdio.h>
int g() ; // my function
// ...
```

```
// my header:
namespace Mine {
   void f() ; // my function
   // ...
}
#include<stdio.h>
namespace Mine {
   int g() ; // my function
   // ...
}
```

Namespaces Are Open

- Defining a previously declared entity of a namespace, it is safer to use the scope resolution operator :: using the syntax *Mine::* rather than re-open the namespace *Mine*

```
namespace Mine {
   void f();
}
void Mine::ff() // error: no ff() declared in Mine
{
   // ...
}
```

- The compiler will not detect this error when trying to redefine *ff()* in the reopened namespace

```
namespace Mine { // reopening Mine to define functions
   void ff() // oops! no ff() declared in Mine;
             // ff() is added to Mine by this definition
   {
        // ...
   }
   // ...
}
```

Namespaces and C

- Declaration of entities defined in files compiled by a C compiler need to be preceded in C++ by a declarator *extern "C"*

```
extern "C" int printf(const char* ... );
```

```
extern "C" {
  ...
  int printf(const char* ... );
  ...
};
```

- C++ defines the set of standard headers, where the C library functions are available in the *std* namespace

```
// cstdio
namespace std {
  // ...
  extern "C" int printf(const char* ... );
  // ...
}
```

```
#include <cstdio>
// ...
  std::printf("Hello, world\n");
// ...
```