CHAPTER 11

SEPARATE COMPILATION

AND

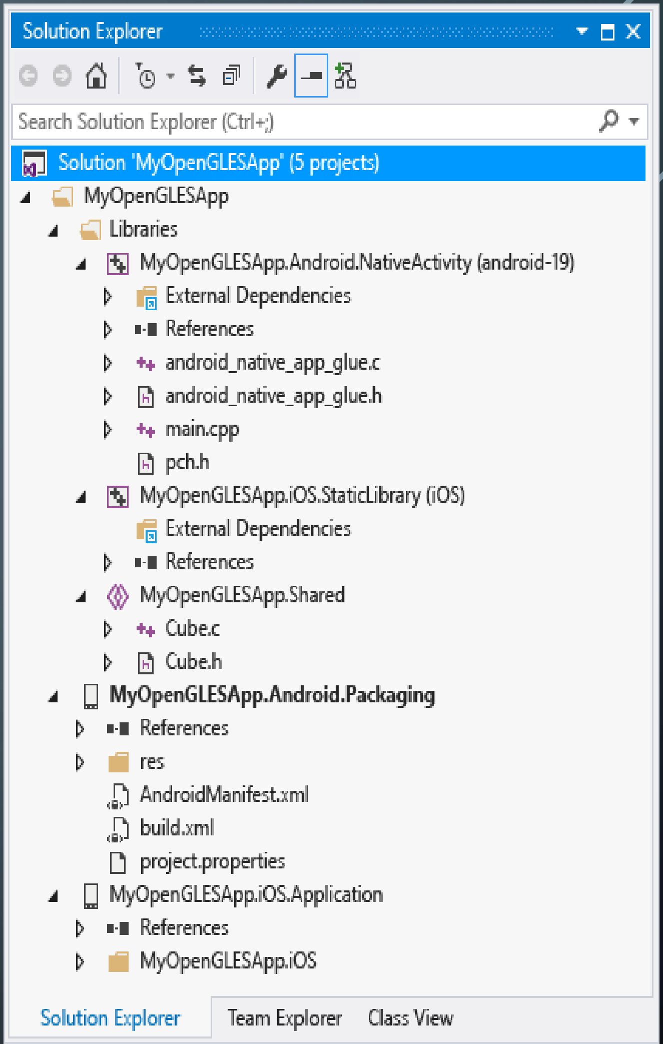
NAMESPACES

LEARNING OBJECTIVES

- Separate Compilation
  - Encapsulation reviewed
  - Header and implementation files
- Namespaces
  - using directives
  - Qualifying names
  - Unnamed namespaces
  - Hiding helping functions
  - Nested namespaces

SEPARATE COMPILATION

- Program Parts
  - Kept in separate files
  - Compiled separately
  - Linked together before program runs
- Class definitions
  - Separate from "using" programs
  - Build library of classes
    - Re-used by many different programs
    - Just like predefined libraries



Solution Explorer

Search Solution Explorer (Ctrl+;)

```
Solution 'MyOpenGLESApp' (5 projects)
◢ MyOpenGLESApp
   ◢ Libraries
      ◢ MyOpenGLESApp.Android.NativeActivity (android-19)
         ▷ External Dependencies
         ▷ References
         ▷ android_native_app_glue.c
         ▷ android_native_app_glue.h
         ▷ main.cpp
           pch.h
      ◢ MyOpenGLESApp.iOS.StaticLibrary (iOS)
           External Dependencies
         ▷ References
      ◢ MyOpenGLESApp.Shared
         ▷ Cube.c
         ▷ Cube.h
   ◢ MyOpenGLESApp.Android.Packaging
      ▷ References
      ▷ res
        AndroidManifest.xml
        build.xml
        project.properties
   ◢ MyOpenGLESApp.iOS.Application
      ▷ References
      ▷ MyOpenGLESApp.iOS
```

CLASS SEPARATION

- Class Independence
  - Separate class definition/specification
    - Called "interface"
  - Separate class implementation
  - Place in two files
- If implementation changes → only that file need be changed
    - Class specification need not change
    - "User" programs need not change

ENCAPSULATION REVIEWED

- Encapsulation principle:
  - Separate how class is used by programmer from details of class's implementation
- "Complete" separation
  - Change to implementation → NO impact on any other programs
- Basic OOP principle

ENCAPSULATION RULES

- Rules to ensure separation:

1. All member variables should be **private**
2. Basic class operations should be:
   - **Public** member functions
   - Friend or ordinary functions
   - Overloaded operators

   Group class definition and prototypes together
   - Called "interface" for class
3. Make class implementation unavailable to users of class

MORE CLASS SEPARATION

- Interface File
  - Contains class definition with function and operator declarations/prototypes
  - Users "see" this
  - Separate compilation unit
- Implementation File
  - Contains member function definitions
  - Separate compilation unit

CLASS HEADER FILES

- Class interface always in header file
  - Use .h naming convention
- Programs that use class will "include" it
  - #include "myclass.h"
  - Quotes indicate you wrote header
    - Find it in **"your"** working directory
  - Recall library includes, e.g., <iostream>
    - < > indicate predefined library header file
    - Find it in library directory

CLASS IMPLEMENTATION FILES

- Class implementation in .cpp file
  - Typically give interface file and implementation file same name
    - myclass.h and myclass.cpp
  - All class's member function defined here
  - Implementation file must #include class's header file
- .cpp files in general, typically contain executable code
  - e.g., Function definitions, including main()

CLASS FILES

- Class header file #included by:
  - Implementation file
  - Program file
    - Often called "application file" or "driver file"
- Organization of files is system dependent
  - Typical IDE has "project" or "workspace"
    - Implementation files "combined" here
    - Header files still "#included"

MULTIPLE COMPILES OF HEADER FILES

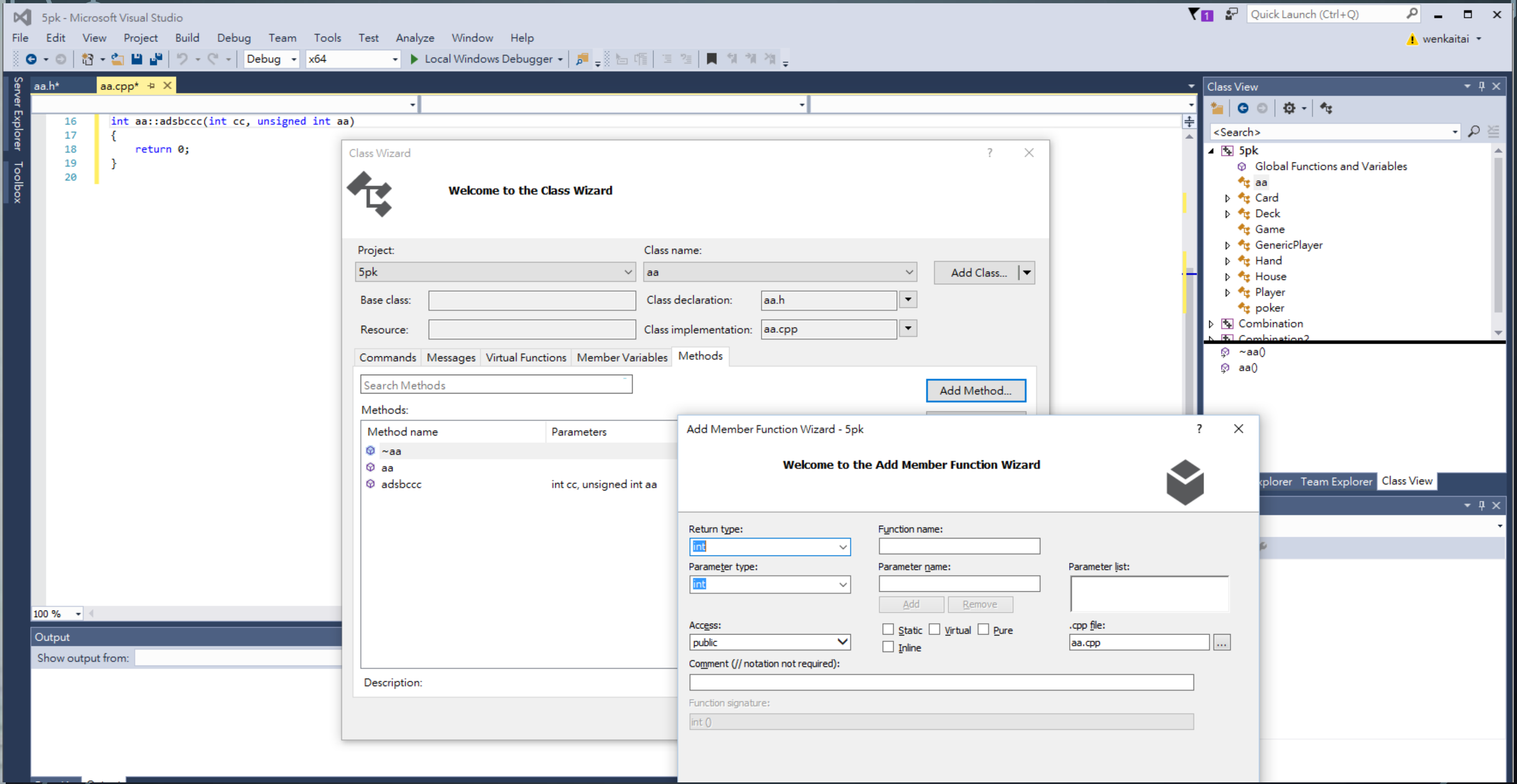
- Header files
  - Typically included multiple times
    - e.g., class interface included by class implementation and program file
  - Must only be compiled once!
  - No guarantee "which #include" in which file, compiler might see first
- Use preprocessor
  - Tell compiler to include header only once

USING #IFNDEF

- Header file structure:

  -
    ```
    #ifndef FNAME_H
    #define FNAME_H
    … //Contents of header file
    …
    #endif
    ```
- FNAME typically name of file for consistency, readability
- This syntax avoids multiple definitions of header file

CLASS WIZARD



5pk - Microsoft Visual Studio

```
int aa::adsbccc(int cc, unsigned int aa)
{
    return 0;
}
```

Class Wizard

Welcome to the Class Wizard

Project: 5pk

Class name: aa

Base class:

Class declaration: aa.h

Resource:

Class implementation: aa.cpp

Methods:

Method name	Parameters
~aa	
aa	
adsbccc	int cc, unsigned int aa

Description:

Add Member Function Wizard - 5pk

Welcome to the Add Member Function Wizard

Return type: int

Function name:

Parameter type: int

Parameter name:

Parameter list:

Access: public

□ Static □ Virtual □ Pure

□ Inline

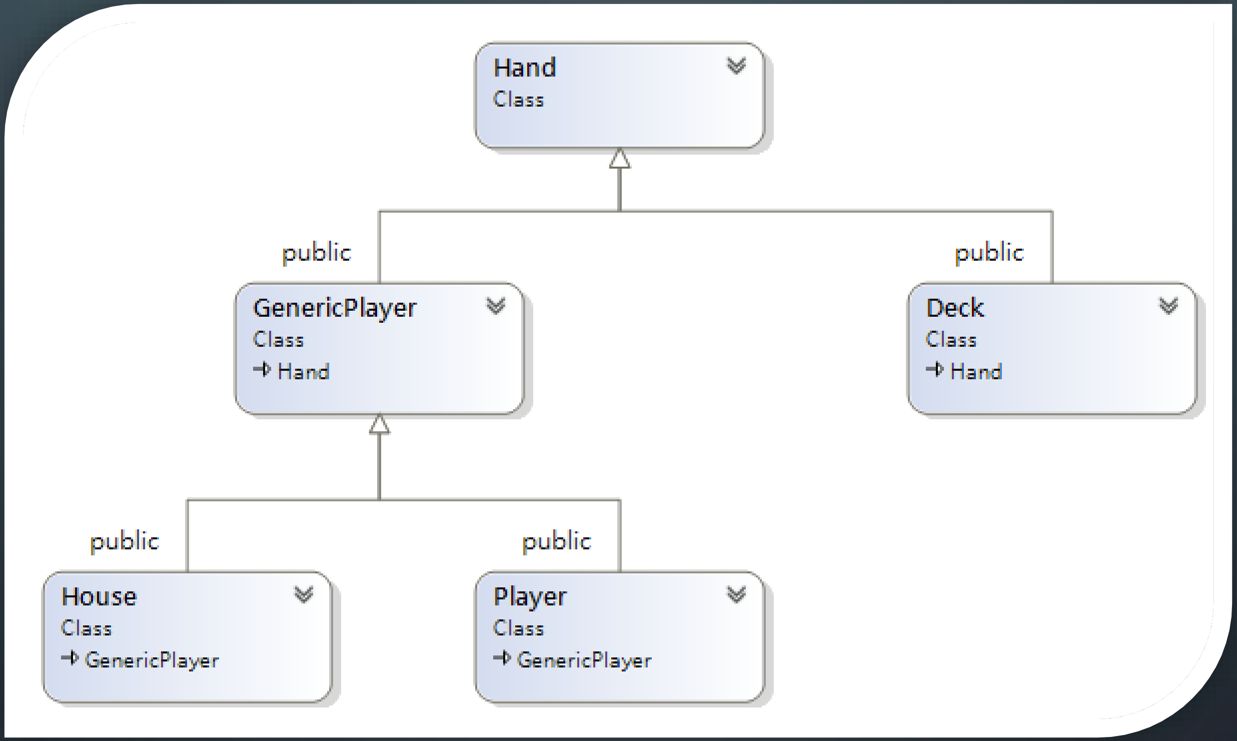
.cpp file: aa.cpp

Comment (// notation not required):

Function signature:

int ()

CLASS DIAGRAM



OTHER LIBRARY FILES

- Libraries not just for classes
- Related functions
  - Prototypes → header file
  - Definitions → implementation file
- Other type definitions
  - structs, simple typedefs → header file
  - Constant declarations → header file

NAMESPACES

- 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 DECLARATION

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

- We've used 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

- Multiple namespaces
  - e.g., global, and std typically used
- 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();
}
```

  - using directive has block-scope

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
```

CREATING A NAMESPACE EXAMPLE

- Function **declaration**:

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

- Function **definition**:

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

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

namespace Space1
{
    void greeting( );
}

namespace Space2
{
    void greeting( );
}

void bigGreeting( );
int main( )
{
    {
        using namespace Space2;
        greeting( );
    }

    {
        using namespace Space1;
        greeting( );
    }

    bigGreeting( );

    return 0;
}
```

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

namespace Space2
{
    void greeting( )
    {
        cout << "Greetings from namespace Space2.\n";
    }
}

void bigGreeting( )
{
    cout << "A Big Global Hello!\n";
}
```

USING DIRECTIVE
USING DECLARATION

USING DECLARATIONS

- Can specify individual names from namespace
- 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;
```

USING DEFINITIONS AND DECLARATIONS

- Differences:
  - using declaration
    - Makes ONE name in namespace *available*
    - Introduces names so no other uses of name are allowed
  - using directive
    - Makes **ALL** names in namespace *available*
    - Only "potentially" introduces names

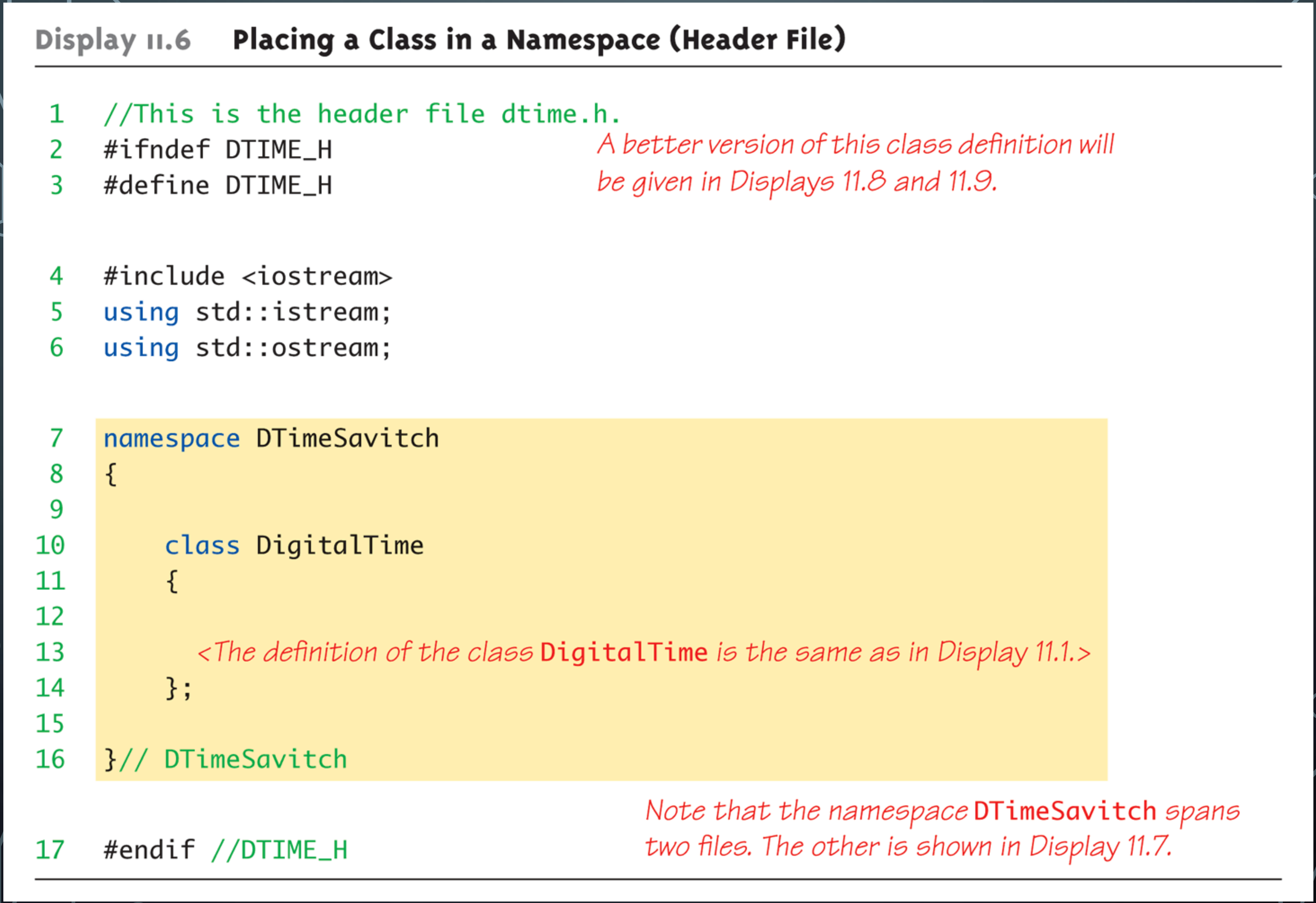
Qualifying NAMES

- Can specify where name comes from
  - Use "qualifier" and scope-resolution operator
  - Used if only intend one use (or few)
- NS1::fun1();
  - Specifies that fun() comes from namespace NS1
- Especially useful for parameters:
  int getInput(std::istream inputStream);
  - Parameter found in istream's std namespace
  - Eliminates need for using directive or declaration

NAMING NAMESPACES

- Include unique string
  - Like last name
- Reduces chance of other namespaces with same name
- Often multiple programmers write namespaces for same program
  - Must have distinct names
  - Without → multiple definitions of same name in same scope
    - Results in error

CLASS NAMESPACE EXAMPLE:



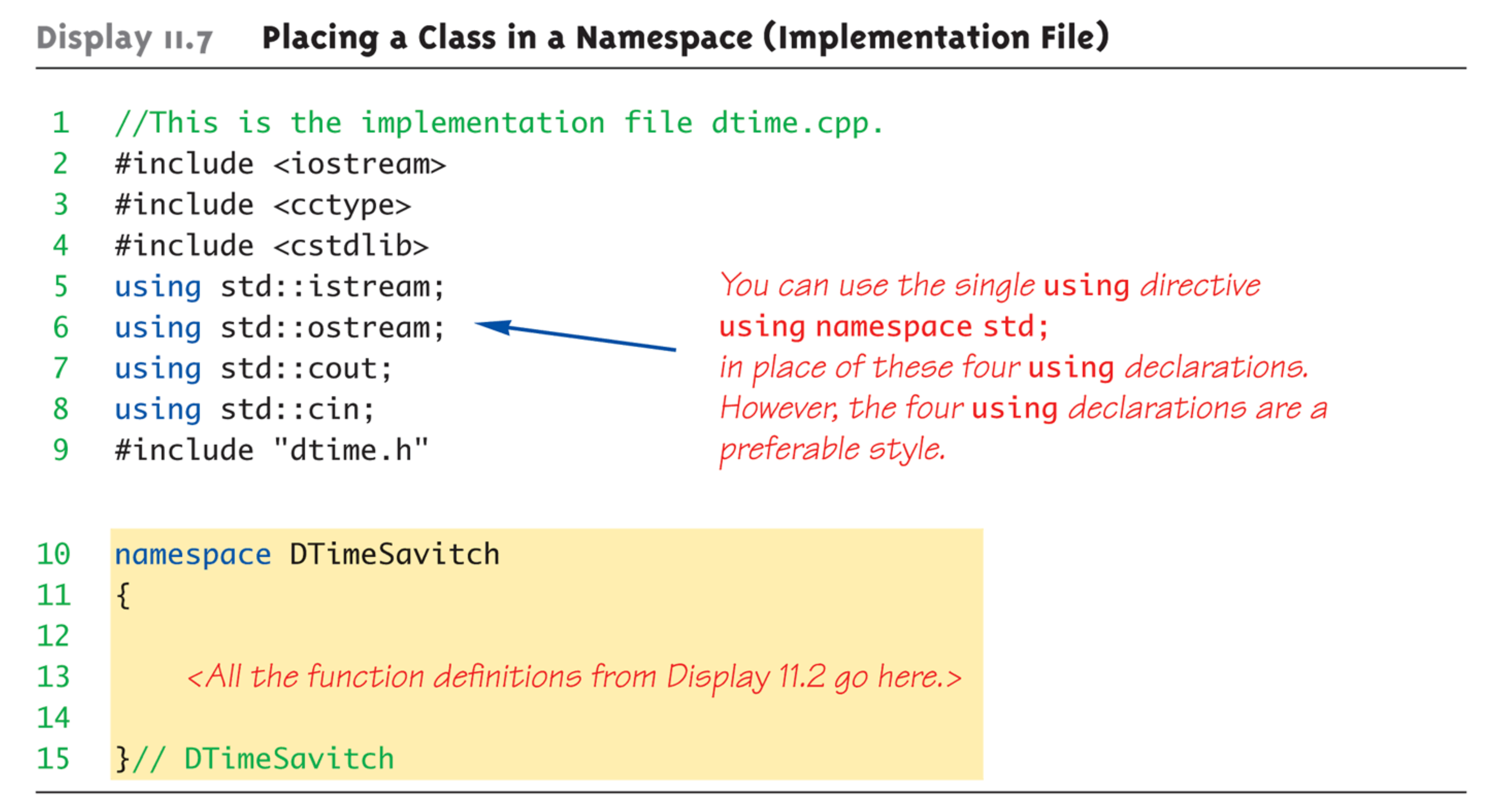
Display 11.6 **Placing a Class in a Namespace (Header File)**

```
//This is the header file dtime.h.
#ifndef DTIME_H
#define DTIME_H

#include <iostream>
using std::istream;
using std::ostream;

namespace DTimeSavitch
{

    class DigitalTime
    {

        <The definition of the class DigitalTime is the same as in Display 11.1.>
    };

}// DTimeSavitch

#endif //DTIME_H
```

A better version of this class definition will be given in Displays 11.8 and 11.9.

Note that the namespace `DTimeSavitch` *spans two files. The other is shown in Display 11.7.*

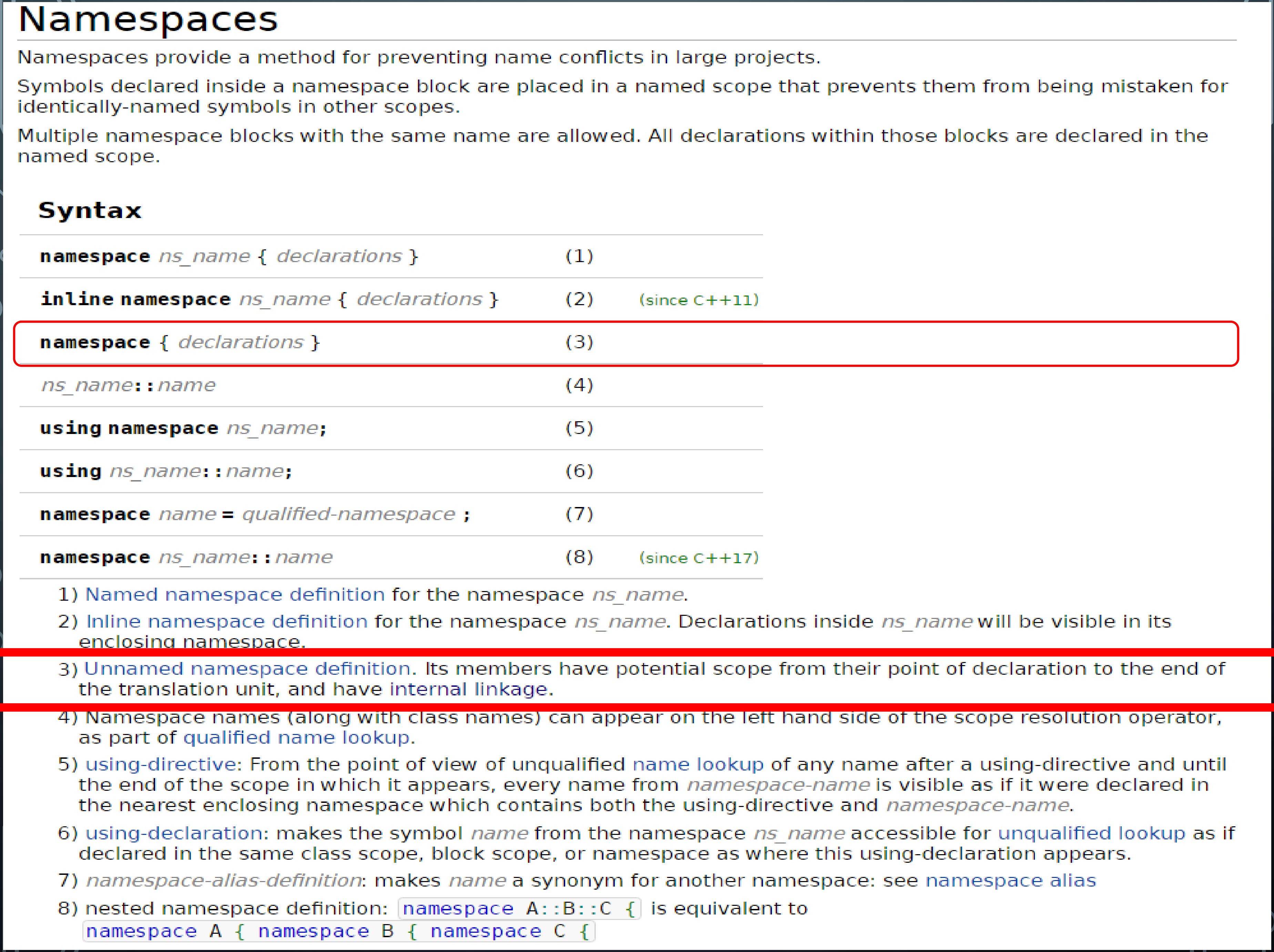
CLASS NAMESPACE EXAMPLE:



Display 11.7 Placing a Class in a Namespace (Implementation File)

```
//This is the implementation file dtime.cpp.
#include <iostream>
#include <cctype>
#include <cstdlib>
using std::istream;
using std::ostream;
using std::cout;
using std::cin;
#include "dtime.h"

namespace DTimeSavitch
{

    <All the function definitions from Display 11.2 go here.>

}// DTimeSavitch
```

You can use the single **using** *directive*
using namespace std;
in place of these four **using** *declarations.*
However, the four **using** *declarations are a preferable style.*



Namespaces

Namespaces provide a method for preventing name conflicts in large projects.

Symbols declared inside a namespace block are placed in a named scope that prevents them from being mistaken for identically-named symbols in other scopes.

Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.

Syntax

namespace *ns_name* { *declarations* }	(1)	
inline namespace *ns_name* { *declarations* }	(2)	(since C++11)
namespace { *declarations* }	(3)	
ns_name::*name*	(4)	
using namespace *ns_name*;	(5)	
using *ns_name*::*name*;	(6)	
namespace *name* = *qualified-namespace* ;	(7)	
namespace *ns_name*::*name*	(8)	(since C++17)

1) Named namespace definition for the namespace *ns_name*.

2) Inline namespace definition for the namespace *ns_name*. Declarations inside *ns_name* will be visible in its enclosing namespace.

3) Unnamed namespace definition. Its members have potential scope from their point of declaration to the end of the translation unit, and have internal linkage.

4) Namespace names (along with class names) can appear on the left hand side of the scope resolution operator, as part of qualified name lookup.

5) using-directive: From the point of view of unqualified name lookup of any name after a using-directive and until the end of the scope in which it appears, every name from *namespace-name* is visible as if it were declared in the nearest enclosing namespace which contains both the using-directive and *namespace-name*.

6) using-declaration: makes the symbol *name* from the namespace *ns_name* accessible for unqualified lookup as if declared in the same class scope, block scope, or namespace as where this using-declaration appears.

7) *namespace-alias-definition*: makes *name* a synonym for another namespace: see namespace alias

8) nested namespace definition: `namespace A::B::C {` is equivalent to `namespace A { namespace B { namespace C {`

NAMESPACE { DECLARATIONS }

- Called anonymous or unnamed namespaces
  - Create an explicit namespace but not give it a name

```
1) namespace
2) {
3)      int MyFunc(){}
4) }
```

- Useful to make variable declarations invisible to code in other files (i.e. give them internal linkage)
  - All code in the same file can see the identifiers in an unnamed namespace but the identifiers are not visible outside that file
  - More precisely outside the translation unit.

UNNAMED NAMESPACES

- Compilation unit defined:
  - A file, along with all files #included in file
- Every compilation unit has unnamed namespace
  - Written same way, but with no name
  - All names are then local to compilation unit
- Use unnamed namespace to keep things "local"
- Scope of unnamed namespace is compilation unit

GLOBAL VS. UNNAMED NAMESPACES

- Not same
- Global namespace:
  - No namespace grouping at all
  - Global scope
- Unnamed namespace:
  - Has namespace grouping, just no name
  - Local scope

Unnamed namespaces as well as all namespaces declared directly or indirectly within an unnamed namespace have internal linkage

Unnamed namespace is with unique name eventually

```
namespace {
    int i;  // defines ::(unique)::i
}
void f() {
    i++;  // increments ::(unique)::i
}

namespace A {
    namespace {
        int i; // A::(unique)::i
        int j; // A::(unique)::j
    }
    void g() { i++; } // A::unique::i++
}

using namespace A; // introduces all names from A into global namespace
void h() {
    i++;    // error: ::(unique)::i and ::A::(unique)::i are both in scope
    A::i++; // ok, increments ::A::(unique)::i
    j++;    // ok, increments ::A::(unique)::j
}
```

NESTED NAMESPACES

- Legal to nest namespaces

```
namespace S1
{
        namespace S2
        {
                void sample()
                {
                        …
                }
        }
```

- Qualify names twice:
  - S1::S2::sample();

```
namespace Q {
  namespace V { // original-namespace-definition for V
    void f(); // declaration of Q::V::f
  }
  void V::f() {} // OK
  void V::g() {} // Error: g() is not yet a member of V
  namespace V { // extension-namespace-definition for V
    void g(); // declaration of Q::V::g
  }
}
namespace R { // not a enclosing namespace for Q
  void Q::V::g() {} // Error: cannot define Q::V::g inside R
}
void Q::V::g() {} // OK: global namespace encloses Q
```

HIDING HELPING FUNCTIONS

- Recall helping function:
  - Low-level utility
  - Not for public use
- Two ways to hide:
  - Make private member function
    - If function naturally takes calling object
  - Place in class implementation’s unnamed namespace!
    - If function needs no calling object
    - Makes cleaner code (no qualifiers)

SUMMARY 1

- Can separate class definition and implementation → separate files
  - Separate compilation units
- Namespace is a collection of name definitions
- Three ways to use name from namespace:
  - Using directive
  - Using declaration
  - Qualifying

SUMMARY 2

- Namespace definitions are placed inside namespace groupings
- Unnamed namespace
  - Used for local name definitions
  - Scope is compilation unit
- Global namespace
  - Items not in a namespace grouping at all
  - Global scope