Unit 6 (Ch 12)

Separate Compilation and Namespaces

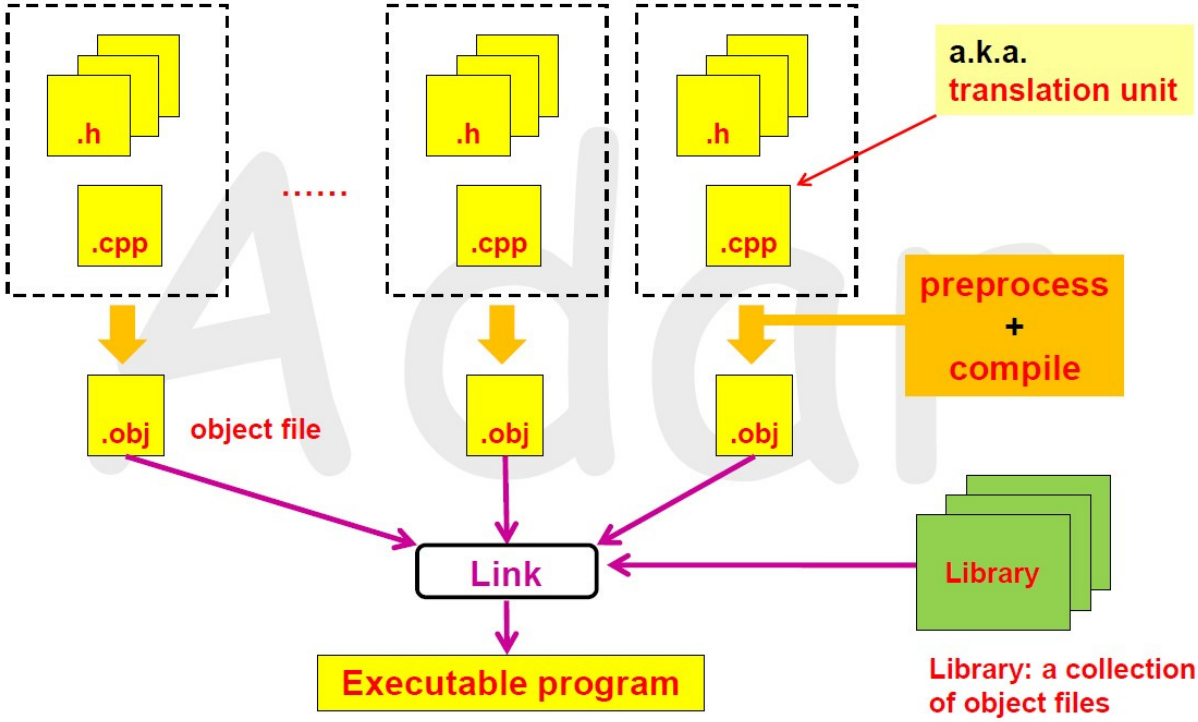
Prof. Chien-Nan (Jimmy) Liu
Dept. of Electronics & Electrical Engr.
Nat'l Yang Ming Chiao Tung Univ.

Tel: (03)5712121 ext:31211
E-mail: jimmyliu@nycu.edu.tw
http://mseda.ee.nctu.edu.tw/jimmyliu

Overview

- *6.1 Separate Complication*
- 6.2 Preprocess & Linkage
- 6.3 Namespaces

Making an Executable Program



Courtesy: Prof. Jiunn-Dar Huang @ NYCUEE

Why Separate Compilation?

- Consider you are doing a software project with million lines of code …
  - Code size of a source file ↑ → compile time ↑
  - Whenever any change (even in .h file) is made, you have to recompile the source file again
- Partition source code based on logical structure
  - Only compile the partial source code with change
  - Better readability and maintainability
- Separate compilation is better for encapsulation
  - Have you ever seen the source code implementing rand() in standard library cstdlib?
  - Promote software reusability

Review ADT in C++

- An ADT is a class defined to separate the interface and the implementation
  - All member variables are private
- The interface of the ADT includes
  - The class definition
  - The declarations of the basic operations, such as
    - Public member functions
    - Friend functions
    - Overloaded operators
- The implementation of the ADT includes
  - The definitions of member functions and operators
  - Make them unavailable to the programmer

Separate Files

- In C++ the ADT interface and implementation can be stored in separate files
  - ADT interface is often stored in a header file (*.h)
  - ADT implementation is often stored in source files (*.cpp)
- C++ allows you to further divide the source code into several .cpp files
  - Each part can be stored and compiled separately
- Class definition and implementation can be stored separately from the main program
  - This allows you to use the class in multiple programs
  - Similar to creating your own libraries of classes

Enable Separate Compilation

- You must provide enough info required to compile a source file in isolation from the rest of the program
  - Ex: before using *cout*, you need to inform compiler in advance by using #include <iostream>
- For user-defined ADT, interface info is provided through the header file, ex: include "dtime.h"
  - Provide definitions of classes and functions only, no implementation details
- C++ does not allow splitting the public and private parts of the class definition across files
  - The entire class definition is usually included in the interface file

Case Study: DigitalTime

- The interface file of the class DigitalTime contains its definition
  - The values of the class are:
    - Time of day, such as 9:30, in 24 hour notation
  - The public members are part of the interface
  - The private members are part of the implementation
- The DigitalTime ADT interface is stored in a file named dtime.h
  - The .h suffix means this is a header file
  - Interface files are always header files
- A program using dtime.h must include it using an include directive → #include "dtime.h"

Example: dtime.h

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

class DigitalTime
{
public:
   friend bool operator ==(const DigitalTime& time1, const DigitalTime& time2);
   DigitalTime(int theHour, int theMinute);
   DigitalTime( );
   void advance(int minutesAdded);
   void advance(int hoursAdded, int minutesAdded);
   friend istream& operator >>(istream& ins, DigitalTime& theObject);
   friend ostream& operator <<(ostream& outs, const DigitalTime& theObject);
private:
   int hour;
   int minute;
};
```

#include " " or < > ?

- To include a predefined header file, use < and >
  Ex: #include <iostream>
  - < and > tells the compiler to look in the system directory that stores predefined header files
- To include a user-defined header file, use " and "
  Ex: #include "dtime.h"
  - " and " usually cause the compiler to look in the current directory for the header file

The Implementation File

- Contains the definitions of the ADT functions
- Usually has the same name as the header file but a different suffix
  - Since our header file is named dtime.h, the implementation file is named dtime.cpp
  - Suffix depends on your system (some use .cxx or .CPP)
- Class implementation also needs to know its definition first
  - Do remember to include the interface file at beginning by #include "dtime.h"

Example: dtime.cpp (1/3)

```
#include <iostream>
#include <cctype>
#include <cstdlib>
#include "dtime.h"
using namespace std;

void readHour(istream& ins, int& theHour);
void readMinute(istream& ins, int& theMinute);
int digitToInt(char c);

bool operator ==(const DigitalTime& time1,
                 const DigitalTime& time2)
{
   return (time1.hour == time2.hour &&
           time1.minute == time2.minute);
}

DigitalTime::DigitalTime( ) : hour(0), minute(0)
{
   //Body intentionally empty.
}
```

```
DigitalTime::DigitalTime(int theHour, int theMinute)
{
   if (theHour < 0 || theHour > 23
       || theMinute < 0 || theMinute > 59)
   {
      cout << "Illegal argument to constructor.";
      exit(1);
   }
   else
   {
      hour = theHour;
      minute = theMinute;
   }
}

int digitToInt(char c)
{
   return ( static_cast<int>(c) –
                          static_cast<int>('0') );
}
```

Example: dtime.cpp (2/3)

```
void DigitalTime::advance(int minutesAdded)
{
    int grossMinutes = minute + minutesAdded;
    minute = grossMinutes%60;
    int hourAdjustment = grossMinutes/60;
    hour = (hour + hourAdjustment)%24;
}

void DigitalTime::advance(int hoursAdded,
                          int minutesAdded)
{
    hour = (hour + hoursAdded)%24;
    advance(minutesAdded);
}

istream& operator >>(istream& ins, DigitalTime&
                                        theObject)
{
    readHour(ins, theObject.hour);
    readMinute(ins, theObject.minute);
    return ins;
}
```

```
ostream& operator <<(ostream& outs, const
                          DigitalTime& theObject)
{
    outs << theObject.hour << ':';
    if (theObject.minute < 10)
        outs << '0';
    outs << theObject.minute;
    return outs;
}

void readMinute(istream& ins, int& theMinute)
{
    char c1, c2;
    ins >> c1 >> c2;

    if (!(isdigit(c1) && isdigit(c2)))
    {
        cout << "Illegal input to readMinute\n";
        exit(1);
    }
```

Example: dtime.cpp (3/3)

```
    theMinute = digitToInt(c1)*10 + digitToInt(c2);
    if (theMinute < 0 || theMinute > 59)
    {
        cout << "Illegal input to readMinute\n";
        exit(1);
    }
}

void readHour(istream& ins, int& theHour)
{
    char c1, c2;
    ins >> c1 >> c2;
    if ( !( isdigit(c1) && (isdigit(c2) || c2 == ':' ) ) )
    {
        cout << "Illegal input to readHour\n";
        exit(1);
    }
```

```
    if (isdigit(c1) && c2 == ':')
    {
        theHour = digitToInt(c1);
    }
    else  //(isdigit(c1) && isdigit(c2))
    {
        theHour = digitToInt(c1)*10 + digitToInt(c2);
        ins >> c2;
        if (c2 != ':')
        {
            cout << "Illegal input to readHour\n";
            exit(1);
        }
    }
    if ( theHour < 0 || theHour > 23 )
    {
        cout << "Illegal input to readHour\n";
        exit(1);
    }
}
```

The Application File

- dtime.cpp is not a complete program yet
  - You have no main() to execute
- Another application file is required, which contains the program that uses the ADT
  - It is also called a driver file
  - Of course, application file also needs to know the class definition before using it → #include "dtime.h"
- In summary, the basic steps to run a program are
  - Compile the implementation file
  - Compile the application file
  - Link the object files to create an executable program
    - The tool, linker, is often done automatically after compilation

Example: test.cpp

```
#include <iostream>
#include "dtime.h"
using namespace std;

int main( )
{
   DigitalTime clock, oldClock;

   cout << "Enter the time in 24-hour notation: ";
   cin >> clock;

   oldClock = clock;
   clock.advance(15);
   if (clock == oldClock)
      cout << "Something is wrong.";
   cout << "You entered " << oldClock
        << endl;
   cout << "15 minutes later the time will be "
        << clock << endl;
   clock.advance(2, 15);
   cout << "2 hours and 15 minutes "
        << "after that " << endl
        << the time will be " << clock
        << endl;

   return 0;
}
```

Sample Dialogue

```
Enter the time in 24-hour notation: 11:15
You entered 11:15
15 minutes later the time will be 11:30
2 hours and 15 minutes after that
the time will be 13:45
```

Why Use Three Separate Files?

- The ADT can be used in other programs without rewriting the definition of the class for each
  - Ensure the consistency between different files
- Implementation file is compiled only once even if multiple programs use the ADT
  - Save compilation time
- Changing the implementation file does not require changing the program using the ADT
  - Hide the implementation details from the users
  - Easier for reuse !!

Overview

- 6.1 Separate Complication
- *6.2 Preprocess & Linkage*
- 6.3 Namespaces

Compile dtime.h ?

- The interface file is not compiled separately
  - Replace any occurrence of *#include "dtime.h"* with the text of dtime.h before compiling
  - This is done at the preprocess step
- Both the implementation file and the application file contain *#include "dtime.h"*
  - The text of dtime.h is seen by the compiler in each of these files
  - There is no need to compile dtime.h separately
- Do remember to put only the declarations that do not need compilation in the header file

Illustration of Preprocessing

dtime.h

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

class DigitalTime
{
   ......
};
```

test.cpp

```
#include <iostream>
#include "dtime.h"
using namespace std;

int main( )
{
   ......
}
```

※ Preprocessor "copies" the texts of dtime.h into test.cpp before compilation

```
#include <iostream>

class DigitalTime
{
   ......
};

using namespace std;

int main( )
{
   ......
}
```

A Header File Usually Contains

- Type definitions — `class XYZ { ..... };`
- Type declarations — `class ABC;`
- Typedefs / type aliases — `typedef intPtr int*;`
- Enumerations — `enum Light {RED, YELLOW, GREEN};`
- Class template definitions — `template<typename T> class V {...};`
- Class template declarations — `template<typename T> class Z;`
- Declarations/definitions of function templates
- Function declarations — `int f(double);`
- Inline function definitions — `inline void g(char ch) { ..... }`
- Data declarations — `extern int a;`
- Constant definitions — `const float PI = 3.14159`
- Include directives — `#include <cstdlib>`
- Macro definitions — `#define VERSION 12`
- Named namespaces — `namespace N { ..... }`

A Header File Should not Contain

- Ordinary function definitions — `char get(char *p) { ..... }`
  - Declarations only
- Data definitions — `int a;`
  - Only constants are allowed
- Aggregate definitions — `short table[] = {1, 2, 3};`
- Unnamed namespaces — `namespace { ... }`
  - Discussed later
- Using directives — `using namespace Foo;`
  - Discussed later
- Unmatched definitions/declarations in different files

Multiple Classes

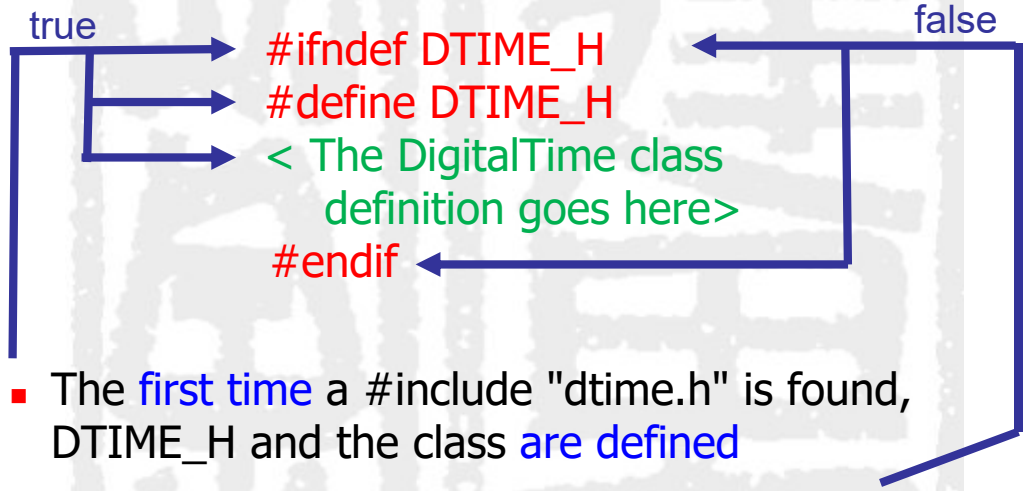
- A program may use several classes
  - Each could be stored in its own interface and implementation files
  - Some files can "include" other files, that include the same definitions in common, ex: <iostream>
- It is possible that the same interface file is included in multiple files
  - C++ does not allow multiple declarations of a class
- The *#ifndef* directive can be used to prevent multiple declarations of a class

Introduction to #ifndef

- To prevent multiple declarations of a class, we can use these directives:
  - #define DTIME_H
    adds DTIME_H to a list indicating DTIME_H is defined
  - #ifndef DTIME_H
    checks to see if DTIME_H has been defined
  - #endif
    If DTIME_H has been defined, skip to #endif
- DTIME_H is the normal convention for creating an identifier to use with ifndef
  - It is the file name in all caps
  - Use ' _ ' instead of ' . '
- Using the same name makes your code easy to read

Using #ifndef

- Consider this code in the interface file



  - The first time a #include "dtime.h" is found, DTIME_H and the class are defined
  - The next time a #include "dtime.h" is found, all lines between #ifndef and #endif are skipped

Ex: Header File with #ifndef

same as in previous example

```
#ifndef DTIME_H
#define DTIME_H
#include <iostream>
using namespace std;

class DigitalTime
{
public:
   friend bool operator ==(const DigitalTime& time1, const DigitalTime& time2);
   DigitalTime(int theHour, int theMinute);
   DigitalTime( );
   void advance(int minutesAdded);
   void advance(int hoursAdded, int minutesAdded);
   friend istream& operator >>(istream& ins, DigitalTime& theObject);
   friend ostream& operator <<(ostream& outs, const DigitalTime& theObject);
private:
   int hour;
   int minute;
};

#endif  //DTIME_H
```

Variable Linkage between Files

- You may use the variables/functions declared at somewhere else, even out of the current file
  - Special treatment is required about their linkage
- Internal linkage
  - A name can be used only in the file in which it is defined
  - Ex: constants, type aliases, …
- External linkage
  - A name can be used in files different from the one in which it is defined
  - A keyword "extern" is required to notify the compiler; otherwise, this variable is undefined in the current scope
- No linkage
  - Local names (names defined inside functions)

Example of External Linkage

- Programmer must ensure that all declarations referring to the same entity are consistent
  - a is ok, defined in file1 and used in file2
  - b is not ok, a global variable cannot be defined twice
  - c is not ok, there is no actual definitions in both files
  - Functions are implicitly declared as extern
    - Using f() is ok with prior declaration

file1.cpp

```
int a = 1;          // Global a
double b = 1.0;     // Global b
extern int c;       // where is it defined?
int f()             // Global f()
{
   int ans;         // no linkage
   ……
}
```

file2.cpp

```
extern int a;   // use a from file1
double b;       // not OK, defined twice
extern int c;   // not OK, still not defined
int f();        // implicitly extern
int g()
{
   int ans = a + f();   // use external
   ……                   // variables and
}                       // functions
```

One-Definition Rule (ODR)

- A class, enum, inline function, template, (and …) must be defined exactly once in a program
  - Ideally, it is defined once in a single file somewhere
  - Just like global variables
- It is difficult for compiler to catch the unmatched definitions in different files
  - Ex: T is int in file1.cpp while T is double in file2.cpp
  - Everything looks good while each file is compiled separately
- How to ensure the definitions are unique and consistent across all files

  → Put all definitions in a header file that is included in all *.cpp
  - If you update the header file, all *.cpp are updated

Overview

- 6.1 Separate Complication
- 6.2 Preprocess & Linkage
- *6.3 Namespaces*

Why Namespace ?

- Assume you get a software library, which contains a function *void common()*, from company ABC
- You also get a software library, which also contains a function *void common()*, from company XYZ
- Once you need to use functions from both libraries in your program, you run into a big trouble
  - Multiple definition for a function → **link error !!**
- Company ABC

```
// in ABC.h
Namespace ABC { void common(); }
```

- Company XYZ

```
// in XYZ.h
Namespace XYZ { void common(); }
```

- In user's program:

```
#include "ABC.h"
#include "XYZ.h"
void func() {
  ABC::common();
  XYZ::common();
}
```


Namespaces

- A **namespace** is a collection of name definitions, such as class definitions and variable declarations
- To place code in a namespace
  - Use a *namespace* grouping
    - namespace Name_Space_Name
      {
        Some_Code
      }
- To use the namespace created → *using* directive
  - Assign namespace for each function → not convenient
    - using Name_Space_Name::function_name();
  - Use all functions → one-time effort
    - using namespace Name_Space_Name;

Namespaces: Adding a Function

- To add a function to a namespace
  - Declare the function in a namespace grouping

```
namespace savitch1
{
    void greeting( );
}
```

- Specify the namespace while defining the function
  - Define the function in a namespace grouping

```
namespace savitch1
{
    void greeting( )
    {
        cout << "Hello from namespace ";
        cout << "savitch1.\n";
    }
}
```

The Using Directive

- #include <iostream> places names such as *cin* and *cout* in the std namespace
- The program does not know about names in the std namespace until you add *using namespace std;*
  - If you do not use the std namespace, you can define cin and cout to behave differently
- To use a function defined in a namespace
  - Include the using directive in the program where the namespace is to be used
  - Call the function as the function would normally be called
  - You can assign a range for the using directive

```
int main( )
{
    {
        using namespace savitch1;
        greeting( );
    }
    .....
```

Example of Namespace (1/2)

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

namespace savitch1
{
   void greeting( );
}

namespace savitch2
{
   void greeting( );
}

void bigGreeting( );
```

```
int main( )
{
   {
      using namespace savitch2;
      greeting( );
   }

   {
      using namespace savitch1;
      greeting( );
   }

   bigGreeting( );

   return 0;
}
```

Execute different functions !!

Example of Namespace (2/2)

```
namespace savitch1
{
   void greeting( )
   {
      cout << "Hello from namespace
              savitch1.\n";
   }
}
```

```
namespace savitch2
{
   void greeting( )
   {
      cout << "Greetings from namespace
              savitch2.\n";
   }
}

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

Sample Dialogue

```
Greetings from namespace savitch2.
Hello from namespace savitch1.
A Big Global Hello!
```

A Namespace Problem

- If the same name is used in two namespaces
  - The namespaces cannot be used at the same time
- Suppose you have the namespaces below:

```
namespace ns1
{
    fun1( );
    myFunction( );
}
```

```
namespace ns2
{
    fun2( );
    myFunction( );
}
```

- Is there an easier way to use both namespaces considering that myFunction is in both?

Qualifying Names

- To select individual functions from namespaces
  - *using ns1::fun1;* //makes only fun1 in ns1 avail
    - The scope resolution operator identifies a namespace here
    - Means we are using **ONLY** namespace ns1's version of fun1
  - If you only want to use the function once, call it like this *ns1::fun1( );*
    - Similar to calling a member function of a class
- To qualify the type of a parameter from namespaces
  - Use the namespace and the type name
    int getNumber (std::istream inputStream)
    ...
    - istream is the istream defined in namespace std
    - If istream is the only name needed from namespace std, then you do not need *using namespace std*

Example of Qualified Names

```
namespace Crystal {
    void func3( );
    void func4( );
}

namespace Stone {
    void func1( );
    void func2( );
}

void Stone::func1( ) {    // in Stone's scope
    func2( );             // call Stone::func2( )
    func3( );             // error!! No func3 in this scope
    Crystal::func3( );    // OK, namespace is specified
}
```

```
// Another alternative way
void Stone::func2( ) {    // in Stone's scope
    using Crystal::func4;
    func4( );             // call Crystal::func4( )
    ......
    func4( );             // call Crystal::func4( )
}
```

It's OK now without specifying the namespace of a function.

Directive/Declaration

- A using directive (e.g: *using namespace std*) makes all the names in the namespace available
- A using declaration (e.g: *using std::cout*) makes only one name available from the namespace
- You can also add using declaration in another namespace
  - Introduce a name from other scope into this one

```
namespace Stone {
    void func1( );
    void func2( );
    using Crystal::func3;
  // func3 now in Stone's scope
}
```

  - Make ALL names from Crystal accessible

```
namespace Stone {
    void func1( );
    void func2( );
    using namespace Crystal;
  // all names in Stone's scope
}
```

A Subtle Point

- A using directive potentially introduces a name
  - If ns1 and ns2 both define myFunction,
    ```
    using namespace ns1;
    using namespace ns2;
    ```
    is OK if myFunction is never used!
- A using declaration introduces a name into your code right away
  - No other use of the name can be made after that
    ```
    using ns1::myFunction;
    using ns2::myFunction;
    ```
    is illegal, even if myFunction is never used

Using Declarations v.s Using Directives

using directive

```
namespace X {
    extern int i, j, k;
}
int k;      // global k in this file

void f( ) {
    int i = 0;
    using namespace X;
    ++i;   // local i
    ++j;   // X::j
    ++k;   // error!! X::k or global k?
    ++::k;     // global k
    ++X::k;   // X's k
}
```

using declaration

```
namespace X {
    extern int i, j, k;
}
int k;      // global k in this file

void f( ) {
    int i = 0;
    using X::i;    // error!! double define
    using X::j;
    using X::k;   // hide global k
    ++j;          // X::j
    ++k;          // X::k, not global k
}
```

All names from X are injected into the namespace of function f.

Unnamed Namespaces

- By default, functions and data variables defined in global scope have external linkage
  - You can link to them if you know their names
- What if you want to keep them local within a file?
  - Ex: hide the implementation of private functions
- Use unnamed namespaces !!
  - No name to be accessed outside the file …
  - Ex:

```
namespace {                 void g( ) {
    int a;                      f( );  // call f() in unnamed namespace
    void f( ) { … };            // ….
    // ….                   }
}
```

Names defined in the unnamed namespace are local to the compilation unit (i.e. this file)

The Unnamed Grouping

- Every compilation unit can have an unnamed namespace
  - Looks like other namespace, but no name is given:

```
namespace
{
    void sample_function( )
    …
} //unnamed namespace
```

- Names in the unnamed namespace
  - Can be used in the compilation unit without a namespace qualifier
  - Cannot be reused outside the compilation unit
- The DigitalTime interface of previous example is rewritten using unnamed namespace as follows

Modified DigitalTime Class

dtime.h

```
#ifndef DTIME_H
#define DTIME_H
#include <iostream>
using namespace std;
class DigitalTime
{ ..... };
#endif   //DTIME_H
```

class definition is also a public information → in named namespace

```
#ifndef DTIME_H
#define DTIME_H
#include <iostream>
using namespace std;
namespace dtimesavitch
{
    class DigitalTime
    { ..... };
}
#endif   //DTIME_H
```

dtime.cpp

```
// include files
void readHour(istream& ins, int& theHour);
void readMinute(istream& ins, int& theMinute);
int digitToInt(char c);
// other friend functions and member functions
```

```
.....
namespace {    // declare unnamed namespace
    void readHour(...);
    void readMinute(...);
    int digitToInt(...);
}
namespace dtimesavitch {
    // friend functions and member functions
}
namespace {
    // private functions for local scope only
}
```

3 local functions are put into unnamed namespace

public functions are put into dtimesavitch namespace

Test.cpp with Namespace

Unnamed namespace is not in this file → define a local version

```
#include <iostream>
#include "dtime.h"
void readHour(int& theHour);

int main( )
{
    using namespace std;
    using namespace dtimesavitch;
    int theHour;
    readHour(theHour);   // execute local version

    DigitalTime clock(theHour, 0), oldClock;
    oldClock = clock;
    clock.advance(15);
    if (clock == oldClock)
        cout << "Something is wrong.";
    cout << "You entered " << oldClock << endl;
    cout << "15 minutes later the time will be "
         << clock << endl;
    clock.advance(2, 15);
    cout << "2 hours and 15 minutes after that\n"
         << "the time will be " << clock << endl;
    return 0;
}

void readHour(int& theHour)   // local version
{
    using namespace std;
    cout << "Let's play a time game.\n"
         << "Let's pretend the hour has just
             changed.\n"
         << "You may write midnight as either
             0 or 24,\n"
         << "but I will always write it as 0.\n"
         << "Enter the hour as a number (0-24): ";
    cin >> theHour;
    if (theHour == 24)
        theHour = 0;
}
```

Compilation Units Overlap

- A header file can be #included in many files
  - If there is an unnamed namespace in the header file, you will have two unnamed namespaces now …
  - This is OK as long as each of the compilation units makes sense independent of the other
  - A name in the header file's unnamed namespace cannot be defined again in other unnamed namespaces
- To avoid choosing a name for a namespace that has already been used
  - Add your last name to the name of the namespace
  - Or, use some other unique string

Global or Unnamed Namespace

- Names in the global namespace have global scope
  - They are available without a qualifier to all program files
  - Each file shares the same global space
- Global and unnamed namespaces are different !!
- Global namespace
  - No name, but implicitly defined
  - Global scope → can be accesses by all files
  - Only one global namespace in a program (shared space)
- Unnamed namespace
  - No name, but explicitly defined
  - Local scope → can only be accessed within a file
  - Each file can have its own unique unnamed namespace