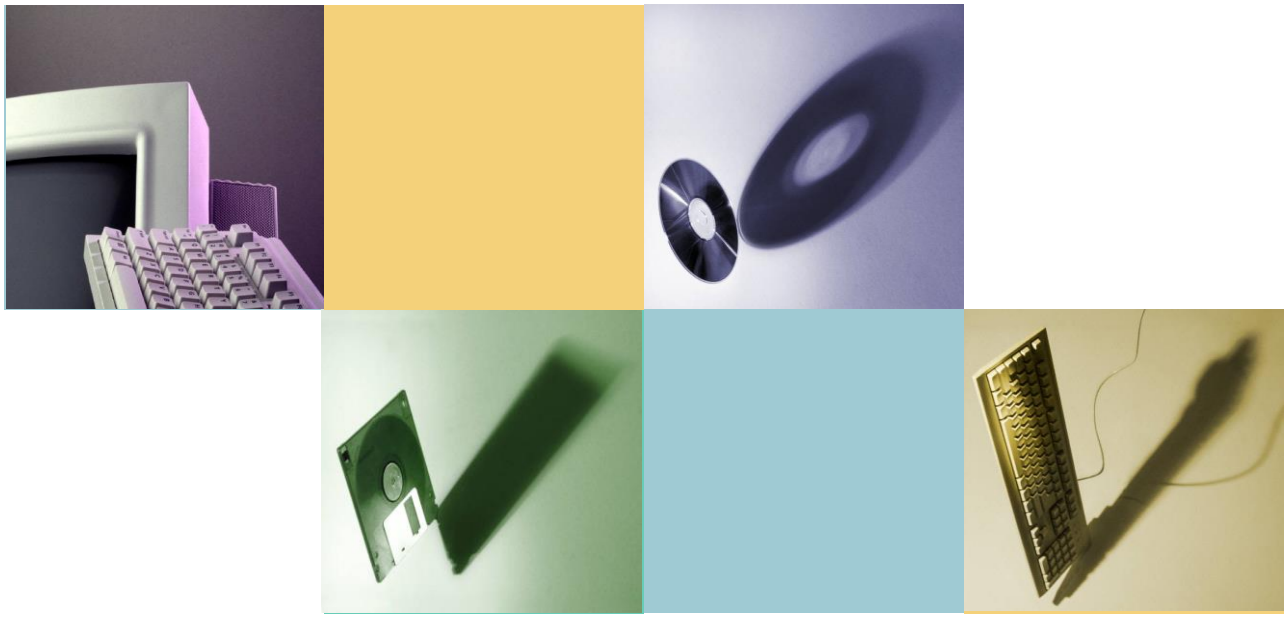
Object-Oriented Programming



Chuan-Kang Ting

Dept of Computer Science and Information Engineering

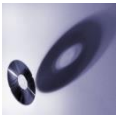
National Chung Cheng University

Chapter 11

Separate Compilation and Namespaces

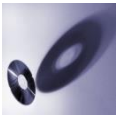
Encapsulated?

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

class DigitalTime
{
  public:
    DigitalTime(int theHour, int theMinute);
    DigitalTime();
    ...
  private:
    int hour;
    ...
};

DigitalTime::DigitalTIme(int theHour, int theMinute)
{
  ...
}
...

int main()
{
  DigitalTime clock, oldClock;
  ...

  return 0;
}
```

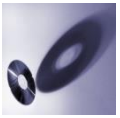

Outline

- **Separate Compilation**
- **Namespaces**


Separate Compilation (1)

- **Program**
  - As you might do in C: divide a program into parts
  - These program parts are
    - Kept in separate files
    - Compiled separately
    - Linked together before program runs
- **With class**
  - **How** and **what** to separate?

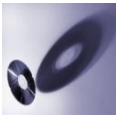




Separate Compilation (2)

- **Class – How and what to separate?**
  - Separate a class *from* using **programs**:

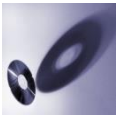
    Build up a library of classes
    - Many programs use the same class
    - Compile once, use it in many programs
    - Just like predefined libraries
      - iostream
      - cstdlib
  - Separate a class *into* **two files**:
    - Specification (interface)
    - Implementation (implementation)

    → Advantages?


Encapsulation Reviewed (1)

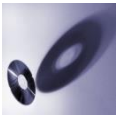
- **Encapsulation**
  - Separate:
    the specification of *how the class is used* by a programmer from the details of *how the class is implemented*
    - Interface: The rule/specification for how to use the class
    - Implementation: The details of how the interface of a class is realized as code
  - "Complete" separation
    - Change the implementation
      → NO need to change any program that uses the class
  - Basic OOP Principle!


Encapsulation Reviewed (2)

- **Ensure the separation**
  1. Make all member variables private
  2. Make each basic operation for a class:
     - Public member function
     - Friend or ordinary function
     - Overloaded operator

     Group *class definition* and *function/operator declaration* together
     - The group is called the **interface** for the class
  3. Make implementation of basic operations unavailable to users of the class
     - Function and overloaded operator *definitions*

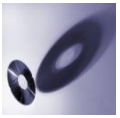




Encapsulation Reviewed (3)

- **The best way to follow the rules**
  - **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

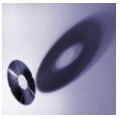
Compilation Unit:

A file, along with all the #included files


Preview Separation

- **Class**
  - Interface file (class header file)
  - Implementation file (class implementation file)
- **Application**
  - Application file

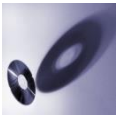

Class Header Files

- **Interface files**
  - Public members
  - Comments
  - Private members (!)
    - are part of implementation, even though in interface file

} ***Interface*** *programmer needs to know*

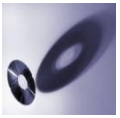
we will fix this weird point later

- **Header files**
  - Interface file is always header file
  - The .h file
    - `#include "myclass.h"`
      - "*custom_header.h*": find it in the working directory
      - <*predefined_header*>: find it in the library directory
    - No need to be compiled

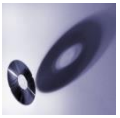

Class Implementation Files

- **Implementation files**
  - in .cpp file
    - The .cpp files generally contain executable code
    - Need to be compiled
  - Typically, give interface file and implementation file the same name
    - Interface: **myclass**.h
    - Implementation: **myclass**.cpp
  - All member functions are defined here
  - Must `#include` class's header file (why?)


Application Files

- **Application files**
  - Also called *driver files*
  - Contain the program (i.e. with the `main` function)
    - → .cpp files
    - Need to be compiled
  - Must **`#include`** class's header file to use the class (why?)

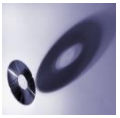

Separation

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

class DigitalTime
{
  public:
    DigitalTime(int theHour, int theMinute);
    DigitalTime();
    ...
  private:
    int hour;
    ...
};

DigitalTime::DigitalTIme(int theHour, int theMinute)
{
  ...
}
...

int main()
{
  DigitalTime clock, oldClock;
  ...

  return 0;
}
```

Interface
(.h)

Implementation
(.cpp)

Application
(.cpp)

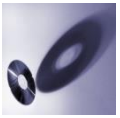




Example – interface

Display 11.1 Interface File for the DigitalTime Class

```
//This is the header file dtime.h. This is the interface for the class DigitalTime.
//Values of this type are times of day. The values are input and output in 24-hour
//notation, as in 9:30 for 9:30 AM and 14:45 for 2:45 PM.
#include <iostream>
using namespace std;

class DigitalTime
{
public:
    DigitalTime(int theHour, int theMinute);
    DigitalTime( );
    //Initializes the time value to 0:00 (which is midnight).

    int getHour( ) const;
    int getMinute( ) const;
    void advance(int minutesAdded);
    //Changes the time to minutesAdded minutes later.

    void advance(int hoursAdded, int minutesAdded);
    //Changes the time to hoursAdded hours plus minutesAdded minutes later.

    friend bool operator ==(const DigitalTime& time1,
                            const DigitalTime& time2);

    friend istream& operator >>(istream& ins, DigitalTime& theObject);

    friend ostream& operator <<(ostream& outs, const DigitalTime& theObject);
private:

    int hour;
    int minute;

    static void readHour(int& theHour);
    //Precondition: Next input to be read from the keyboard is
    //a time in notation, like 9:45 or 14:45.
    //Postcondition: theHour has been set to the hour part of the time.
    //The colon has been discarded and the next input to be read is the minute.

    static void readMinute(int& theMinute);
    //Reads the minute from the keyboard after readHour has read the hour.

    static int digitToInt(char c);
    //Precondition: c is one of the digits '0' through '9'.
    //Returns the integer for the digit; for example, digitToInt('3') returns 3.

};
```

These member variables and helping functions are part of the implementation. They are not part of the interface. The word **private** *indicates that they are not part of the public interface.*


Example – implementation (1)

Display 11.2 Implementation File (part 1 of 3)

```
//This is the implementation file dtime.cpp of the class DigitalTime.
//The interface for the class DigitalTime is in the header file dtime.h.
#include <iostream>
#include <cctype>
#include <cstdlib>
using namespace std;
#include "dtime.h"

//Uses iostream and cstdlib:
DigitalTime::DigitalTime(int theHour, int theMinute)
{
    if (theHour < 0 || theHour > 24 || theMinute < 0 || theMinute > 59)
    {
        cout << "Illegal argument to DigitalTime constructor.";
        exit(1);
    }
    else
    {
        hour = theHour;
        minute = theMinute;
    }

    if (hour == 24)
        hour = 0; //Standardize midnight as 0:00
}

DigitalTime::DigitalTime( )
{
    hour = 0;
    minute = 0;
}

int DigitalTime::getHour( ) const
{
    return hour;
}

int DigitalTime::getMinute( ) const
{
    return minute;
}

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

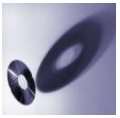
Display 11.2 Implementation File (part 2 of 3)

```
    hour = (hour + hourAdjustment)%24;
}

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

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

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

//Uses iostream:
istream& operator >>(istream& ins, DigitalTime& theObject)
{
    DigitalTime::readHour(theObject.hour);
    DigitalTime::readMinute(theObject.minute);
    return ins;
}

int DigitalTime::digitToInt(char c)
{
    return ( static_cast<int>(c) - static_cast<int>('0') );
}

//Uses iostream, cctype, and cstdlib:
void DigitalTime::readMinute(int& theMinute)
{
    char c1, c2;
    cin >> c1 >> c2;

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

Example – implementation (2)

Display 11.2 Implementation File (part 3 of 3)

```
    theMinute = digitToInt(c1)*10 + digitToInt(c2);

    if (theMinute < 0 || theMinute > 59)
    {
        cout << "Error: illegal input to readMinute\n";
        exit(1);
    }
}

//Uses iostream, cctype, and cstdlib:
void DigitalTime::readHour(int& theHour)
{
    char c1, c2;
    cin >> c1 >> c2;
    if ( !( isdigit(c1) && (isdigit(c2) || c2 == ':' ) ) )
    {
        cout << "Error: illegal input to readHour\n";
        exit(1);
    }

    if (isdigit(c1) && c2 == ':')
    {
        theHour = DigitalTime::digitToInt(c1);
    }
    else //(isdigit(c1) && isdigit(c2))
    {
        theHour = DigitalTime::digitToInt(c1)*10
                  + DigitalTime::digitToInt(c2);
        cin >> c2; //discard ':'
        if (c2 != ':')
        {
            cout << "Error: illegal input to readHour\n";
            exit(1);
        }
    }

    if (theHour == 24)
        theHour = 0; //Standardize midnight as 0:00

    if ( theHour < 0 || theHour > 23 )
    {
        cout << "Error: illegal input to readHour\n";
        exit(1);
    }
}
```

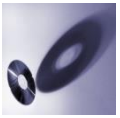

Example – application

Display 11.3 Application File Using `DigitalTime` Class

```
//This is the application file timedemo.cpp, which demonstrates use of DigitalTime
#include <iostream>
using namespace std;
#include "dtime.h"

int main( )
{
    DigitalTime clock, oldClock;

    cout << "You may write midnight as either 0:00 or 24:00,\n"
         << "but I will always write it as 0:00.\n"
         << "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\n"
         << "the time will be "
         << clock << endl;

    return 0;
}
```

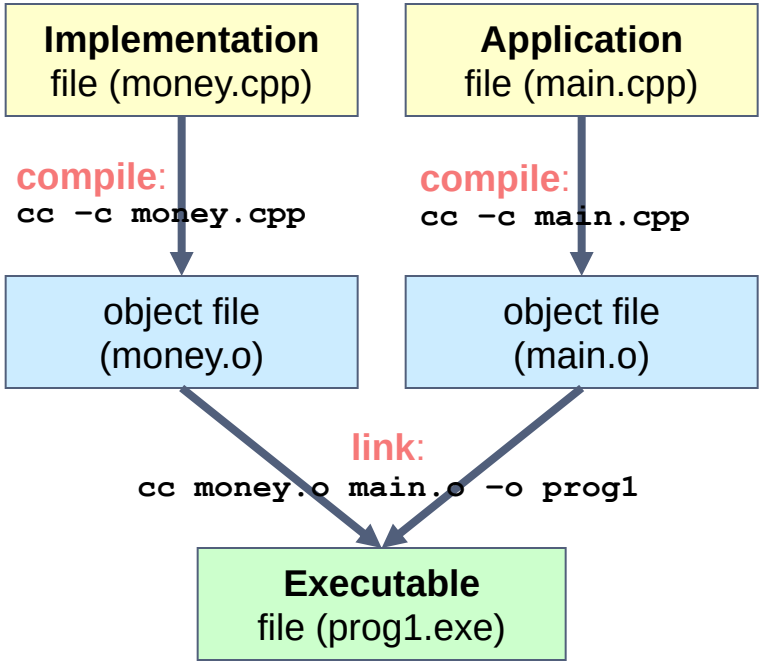
Sample Dialogue

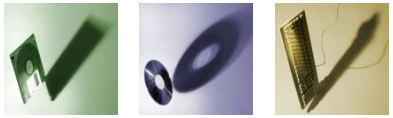
```
You may write midnight as either 0:00 or 24:00,
but I will always write it as 0:00.
Enter the time in 24-hour notation: 11:15
You entered 11:15
15 minutes later the time will be 11:30
2 hours and 15 minutes after that
the time will be 13:45
```


Organizing The Three Files

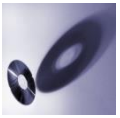
- **Linker**
  - Links the *separately-compiled* implementation file and application file
  - Is system-dependent
- **IDE**
  - Integrated Development Environments
  - Facilitates the compilation process
  - May combine various files into a **project**





Using `#ifndef` (1)

- **Header files**
  - Typically included multiple times
    - e.g., class interface included by class implementation and program file
  - NOT allowed to be compiled more than once
  - Mess in multiple #include
    - No guarantee "which #include" in which file compiler might see first
- **Use preprocessor**
  - Tell compiler to include header only once


Using `#ifndef` (2)

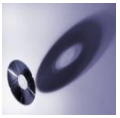
- **Header file structure**
  - Use `#ifndef` to avoid multiple definitions of header file

```
#ifndef FNAME_H
#define FNAME_H



...     //Contents of header file



#endif     //FNAME_H
```

if `FNAME_H` is not defined (seen), then…(do yellow block) until `#endif`

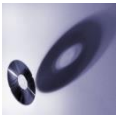
Put `FNAME_H` on a list to indicate "`FNAME_H` *has been seen by compiler*"

`FNAME_H`: typically, the file name
- all uppercase letters
- underscore _ for "."

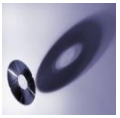

Using `#ifndef` (3)

- **Defining other libraries**
  - Libraries not just for classes
  - Libraries of functions
    - Declarations (prototypes) → header file
    - Definitions → implementation file
  - Other type definitions in header file
    - structs
    - Simple typedefs
    - Constant declarations


Namespaces (1)

- **Namespace**
  - **A collection of *name* definitions**
    - Class definitions
    - Variable declarations
  - Deals with the problem that two classes or functions have the same name
    - Occurs because a programs uses different classes and functions written by different programmers
    - Confusing, even an error for compiler
  - Can be turned "on" or "off"
    - If names might conflict → turn off

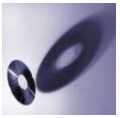




Namespaces (2)

- **`using` directive**
  - We have already used it:

```
using namespace std;
```

    - Makes all definitions in std namespace available (turn on)
    - If not include it, you make `cout` and `cin` have non-standard meaning
      - their standard meaning is in the std namespace
      - could be intended when you want to redefine them
      - thus, the only definitions of `cin` and `cout` the program knows are whatever definitions you give them

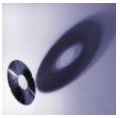




Namespaces (3)

- **Namespace std**
  - Contains all names defined in many standard library files
  - Example:

```
#include <iostream>
```

    - The library places all name definitions (cin, cout, etc.) into **std** namespace
    - Program doesn't know names in the std namespace, unless we specify it → `using namespace std`

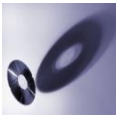




Namespaces (4)

- **Global namespace**
  - Every bit of code you write is in some namespace
    - Specify it
    - If unspecified → global namespace

    } *Either-or*
  - No need for `using` directive
  - Global namespace always available

→ You could say there is *always* an implicit automatic `using` directive that says you are using the *global* namespace

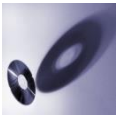




Namespaces (5)

- **Multiple namespaces**
  - You can use more than one namespace in the same program
    - e.g. we are always using *global* and usually using *std*
    - However, each name entry belongs to only one namespace
  - What if **the same name** is **defined in two namespaces**?
    - Error, if you are using both namespaces (name conflict)
    - Choose one → You can only use one of them at a time
      - Can use them each *at different times* in the same program
      - e.g. suppose myFunction() is defined in both NS1 and NS2

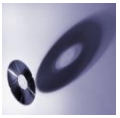
```
{
  using namespace NS1;
  myFunction();
}
{
  using namespace NS2;
  myFunction();
}
```

`using` directive has **block**-scope


Namespaces (6)

- **Scope**
  - A *block* is code enclosed in braces { }
  - The scope of a `using` directive runs from its occurrence to the end of the block
    - A `using` directive is in effect only in its scope
  - If place a `using` directive at the start of a file (outside all blocks), then the directive applies to the entire file


Namespaces (7)

- **Creating a namespace**
  - Use namespace grouping

```
namespace Namespace_Name
{
   //Some_Code
}
```

    - The above puts all names defined in `Some_Code` into namespace *`Namespace_Name`*
  - The defined names can be made available with

```
using namespace Namespace_Name;
```

Namespaces (8)

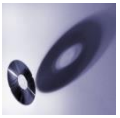
- **Creating a namespace** (cont'd)
  - Can have any number of namespace groupings for a single namespace
    - e.g. function declaration

```
namespace NS1
{
  void greeting();
}
```

function definition

```
namespace NS1
{
  void greeting()
  {
    cout << "Wie geht's? \n";
  }
}
```

Two namespace groupings for namespace NS1


Example

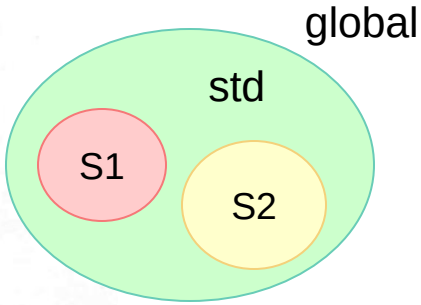
```
#include <iostream>
using namespace std;

namespace Space1
{
    void greeting( );
}

namespace Space2
{
    void greeting( );
}

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

    {
        using namespace Space1;
        greeting( );
    }

    bigGreeting( );

    return 0;
}
```

Names in this block use definitions in namespaces Space2, std, and the global namespace.

Names in this block use definitions in namespaces Space1, std, and the global namespace.

Names out here only use definitions in the namespace std and the global namespace.

map of **using** names



Each name entry belongs to only one namespace

```
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";
}
```

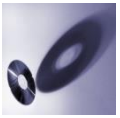
in *Space1*

in *Space2*

in *global*

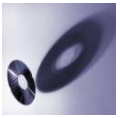
SAMPLE DIALOGUE

```
Greetings from namespace Space2.
Hello from namespace Space1.
A Big Global Hello!
```


Specifying Namespace

- **Three ways**
  - **`using`** directives
  - **`using`** declaration
  - qualifying names


`using` Declarations

- **`using` declaration**
  - Can specify a single name from namespace
  - Syntax:

```
using Name_Space::One_Name;
```

Scope resolution op (::)

Two uses, but similar:

- specify the class for a member function definition
- specify the namespace for a function definition

  - Why bother?
    - e.g. To use func1 and func2

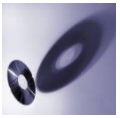
```
using namespace NS1;
using namespace NS2;
```

→ Potential conflict for `myFunc()`

Solution:

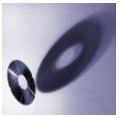
```
using NS1::func1;
using NS2::func2;
```

```
namespace NS1 {
  void func1();
  void myFunc();
}
namespace NS2 {
  void func2();
  void myFunc();
}
```

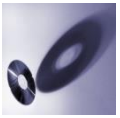

`using` Directives and Declarations

- **`using` declaration**
  - e.g. `using std::cout;`
  - Makes ONE name in namespace available
  - Introduces names so no other uses of name are allowed
- **`using` directive**
  - e.g. `using namespace std;`
  - Makes ALL names in namespace available
  - Only "potentially" introduces names
    - Introduce whenever needed
    - No problem if the conflicting function `myFunc` is never used


Qualifying Names (1)

- **Qualifying names**
  - A way to specify where name comes from
  - Use namespace and scope resolution operator
  - Used if only intend one use (or few)
    - could make code messy


Qualifying Names (2)

- **Specifying functions**

```
NS1::fun1();
```

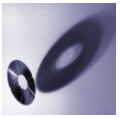
  - Specifies that fun1( ) comes from namespace NS1

- **Specifying parameters**

```
int getInput(std::istream ins);
```

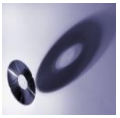
  - Parameter found in istream's std namespace
  - Eliminates need for using directive or declaration
  - Q: Are p1 and p2 the same type?

```
using namespace NS1;
void someFunc(istream p1, std::istream p2);
```


Naming Namespace

- **Name for a namespace**
  - Be unique
    - Reduce the chance of the same namespace name, which may result in errors
    - Important when multiple programmers write code for the same project
  - A good idea to include
    - Your last name (e.g. MoneyTING)
    - Unique string

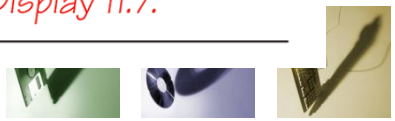

Example – Interface

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.*

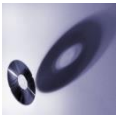


Example – Implementation

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

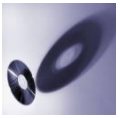
```
//This is the implementation file dtime.cpp.
#include <iostream>
#include <cctype>
#include <cstdlib>
using std::istream;
using std::ostream;
using std::cout;
using std::cin;
#include "dtime.h"

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.*


Unnamed Namespaces (1)

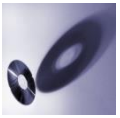
- **Compilation unit**
  - A file, along with all the files #included in the file
    - e.g. a class implementation file, along with the interface header file for the class


Unnamed Namespaces (2)

- **Unnamed namespace**
  - A namespace grouping with NO name
  - Every compilation unit has an unnamed space
  - Makes names local
    - All named defined in unnamed namespace are local to the compilation unit
      - The names can then be **reused outside** compilation unit
    - Any name defined in it can be used ***without*** qualification anywhere in the compilation unit
      - Actually, NO name for qualification
  - Useful for hiding the helping functions

```
namespace
{
  void func1();
  ...
}
```


Example – Hiding Helping Functions (1)

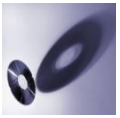
Display 11.8 Hiding the Helping Functions in a Namespace (Interface File)

```
//This is the header file dtime.h. This is the interface for the class DigitalTime.
//Values of this type are times of day. The values are input and output in 24-hour
//notation, as in 9:30 for 9:30 AM and 14:45 for 2:45 PM.
#ifndef DTIME_H
#define DTIME_H

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

namespace DTimeSavitch
{
    class DigitalTime
    {
    public:
        DigitalTime(int theHour, int theMinute);

        DigitalTime( );
        //Initializes the time value to 0:00 (which is midnight).

        getHour( ) const;
        getMinute( ) const;

        void advance(int minutesAdded);
        //Changes the time to minutesAdded minutes later.
        void advance(int hoursAdded, int minutesAdded);
        //Changes the time to hoursAdded hours plus minutesAdded minutes later.

        friend bool operator ==(const DigitalTime& time1,
                                const DigitalTime& time2);
        friend istream& operator >>(istream& ins, DigitalTime& theObject);
        friend ostream& operator <<(ostream& outs,
                                    const DigitalTime& theObject);
    private:
        int hour;
        int minute;
    };

} //DTimeSavitch
#endif //DTIME_H
```

This is our final version of the class **DigitalTime**. *This is the best version and the one you should use.*
The implementation to use with this interface is given in Display 11.9.

Note that the helping functions are not mentioned in the interface file.

```
void readHour(int&);
void readMinute(int&);
void digitToInt(char);
```


Example – Hiding Helping Functions (2)

Display 11.9 Hiding the Helping Functions in a Namespace (Implementation File) (

```
//This is the implementation file dtime.cpp of the class DigitalTime.
//The interface for the class DigitalTime is in the header file dtime.h.
#include <iostream>
#include <cctype>
#include <cstdlib>
using std::istream;
using std::ostream;
using std::cout;
using std::cin;
#include "dtime.h"

namespace
{
    int digitToInt(char c)
    {
        return ( int(c) - int('0') );
    }

    //Uses iostream, cctype, and cstdlib:
    void readMinute(int& theMinute)
    {
        char c1, c2;
        cin >> c1 >> c2;
        if (!(isdigit(c1) && isdigit(c2)))
        {
            cout << "Error: illegal input to readMinute\n";
            exit(1);
        }

        theMinute = digitToInt(c1)*10 + digitToInt(c2);

        if (theMinute < 0 || theMinute > 59)
        {
            cout << "Error: illegal input to readMinute\n";
            exit(1);
        }
    }

//Uses iostream, cctype, and cstdlib:
    void readHour(int& theHour)
    {
        char c1, c2;
        cin >> c1 >> c2;
        if ( !( isdigit(c1) && (isdigit(c2) || c2 == ':' ) ) )
        {
            cout << "Error: 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);
            cin >> c2; //discard ':'
            if (c2 != ':')
            {
                cout << "Error: illegal input to readHour\n";
                exit(1);
            }
        }

        if (theHour == 24)
            theHour = 0; //Standardize midnight as 0:00.

        if ( theHour < 0 || theHour > 23 )
        {
            cout << "Error: illegal input to readHour\n";
            exit(1);
        }
    }
} //unnamed namespace

namespace DTimeSavitch
{
    //Uses iostream:
    istream& operator >>(istream& ins, DigitalTime& theObject)
    {
        readHour(theObject.hour);
        readMinute(theObject.minute);
        return ins;
    }
```

Specifies the unnamed namespace

Names defined in the unnamed namespace are local to the compilation unit. So, these helping functions are local to the file dtime.cpp.

Within the compilation unit (in this case dtime.cpp), you can use names in the unnamed namespace without qualification.

Example – Hiding Helping Functions (3)

Display 11.10 Hiding the Helping Functions in a Namespace (Application Program) (part 1 of 2)

```
//This is the application file timedemo.cpp. This program
//demonstrates hiding the helping functions in an unnamed namespace.

#include <iostream>
#include "dtime.h"

void readHour(int& theHour);

int main( )
{
    using std::cout;
    using std::cin;
    using std::endl;

    using DTimeSavitch::DigitalTime;

    int theHour;
    readHour(theHour);

    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)
{
    using std::cout;
    using std::cin;

    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 to 24): ";
    cin >> theHour;
}
```

If you place the **using** *declarations here, then the program behavior will be the same. (However, many authorities say that you should make the scope of each* **using** *declaration or* **using** *directive as small as is reasonable, and we wanted to give you an example of that technique.)*

This is a different function **readHour** *than the one in the implementation file* **dtime.cpp** *(shown in Display 11.9).*

To use the class in namespace DTimeSavitch

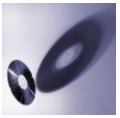
When we gave these **using** *declarations before, they were in* **main**, *so their scope was* **main**. *Thus, we need to repeat them here in order to use* **cin** *and* **cout** *in* **readHour**.

Sample Dialogue

```
Let's play a time game.
Let's pretend the hour has just changed.
You may write midnight as either 0 or 24,
but, I will always write it as 0.
Enter the hour as a number (0 to 24): 11
You entered 11:00
15 minutes later the time will be 11:15
2 hours and 15 minutes after that
the time will be 13:30
```

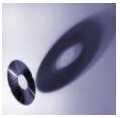
Unnamed Namespaces (3)

- Global and unnamed namespaces are different
- **Global namespaces**
  - No namespace
  - Global scope (to all the program files)
- **Unnamed namespaces**
  - Has namespace grouping, just no name
  - Local scope (to compilation unit)


Hiding Helping Functions

- **Helping functions**
  - Low-level utility
  - Not for public use
- **Two ways to hide:**
  - Make private member function (in header file)
    - If function naturally takes calling object
  - Place in class implementation's unnamed namespace (in implementation file)
    - If function needs no calling object
    - Makes cleaner code, due to no qualifiers

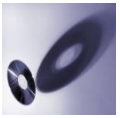




Nested Namespaces

- **Legal to nest namespaces**
  - e.g.

```
namespace S1
{
  namespace S2
  {
    void someFunc();
    ...
  }
}
```

  - Qualify names multiple times
    - `S1::S2::someFunc();` //used outside S1
    - `S2::someFunc();` //used outside S2, in S1


Which Namespace Specification?

- **Three ways to specify**
  - **`using`** directive

```
using namespace theSpace;
```

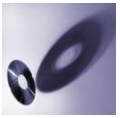
  - **`using`** declaration

```
using theSpace::f;
```

  - Qualifying: omit `using`, but always qualify by

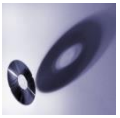
```
theSpace::f()
```

- **The 2nd form is preferred**
  - Omit the unused names in namespace
    - Avoids potential name conflicts
  - Nicely documents which names you use
    - NOT as messy as the 3rd form


Summary (1)

- **Separate Compilation**
  - Separate program into three files
    - Interface file: class definition and comments
    - Implementation file: member function definition
    - Application file: program
  - Encapsulation
  - Use `#ifndef` to deal with multiple compilations


Summary (2)

- **Namespaces**
  - A collection of name definitions
  - Three ways to use name from namespace:
    - **`using`** directive: makes All names available
    - **`using`** declaration: makes One name available
    - Qualifying the name with name of namespace and ::
  - Unnamed namespace can make a name definition local to a compilation unit
    - Useful for hiding helping functions