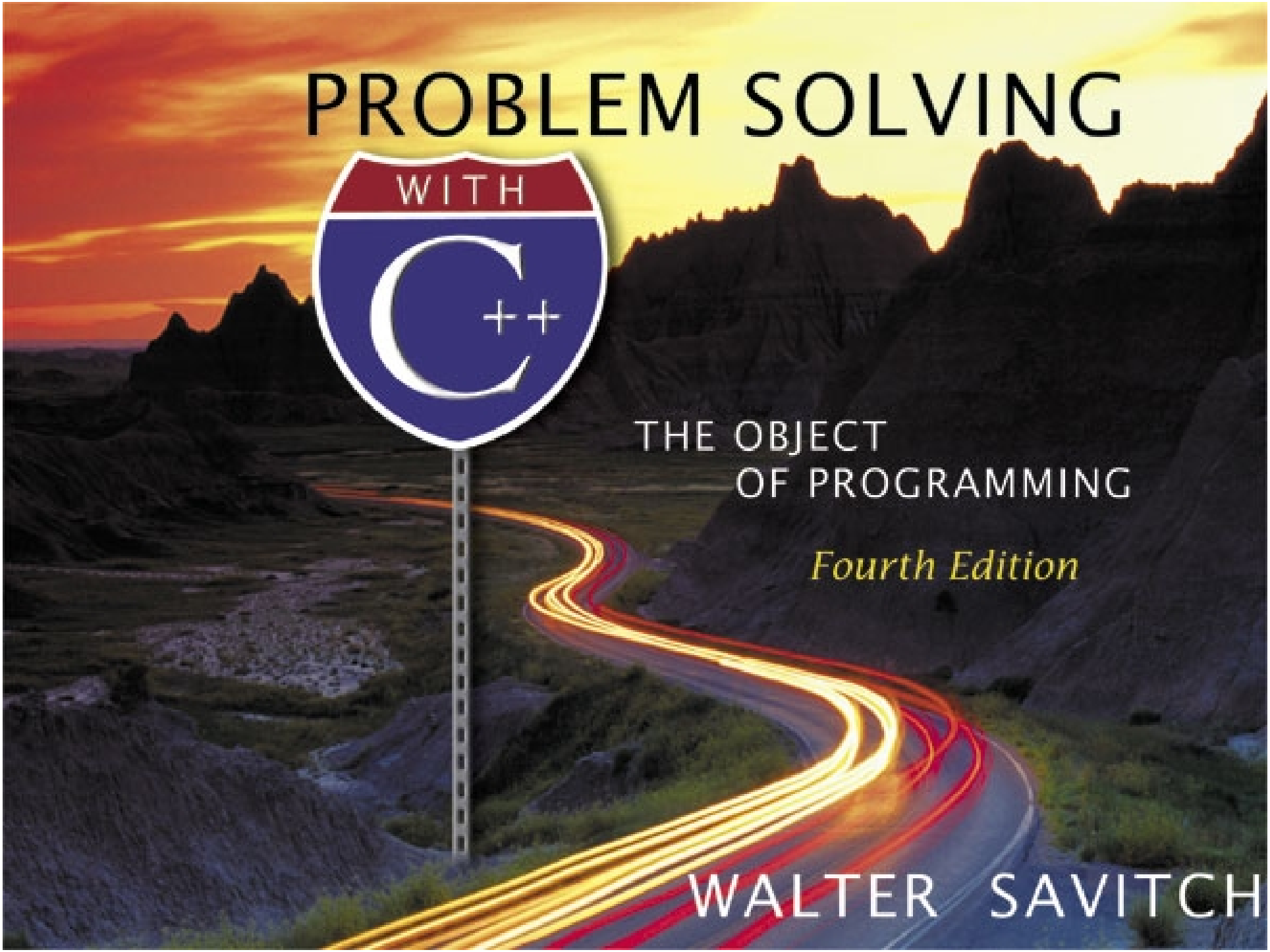
OBJECT ORIENTED PROGRAMMING USING C++


PROBLEM SOLVING
WITH
C++
THE OBJECT
OF PROGRAMMING
Fourth Edition
WALTER SAVITCH

Chapter 9

Separate Compilation and Namespaces

Created by David Mann, North Idaho College

Overview

- Separate Compilation (9.1)
- Namespaces (9.2)

Separate Compilation

- C++ allows you to divide a program into parts
  - Each part can be stored in a separate file
  - Each part can be compiled separately
  - A class definition can be stored separately from a program.
    - This allows you to use the class in multiple programs

ADT Review

- An ADT is a class defined to separate the interface and the implementation
  - All member variables are private
  - The class definition along with the function and operator declarations are grouped together as the interface of the ADT
  - Group the implementation of the operations together and make them unavailable to the programmer using the ADT

The ADT Interface

- The **interface** of the ADT includes
  - The class definition
  - The declarations of the basic operations which can be one of the following
    - Public member functions
    - Friend functions
    - Ordinary functions
    - Overloaded operators
  - The function comments

The ADT Implementation

- The **implementation** of the ADT includes
  - The function definitions
    - The public member functions
    - The private member functions
    - Non-member functions
    - Private helper functions
  - Overloaded operator definitions
  - Member variables
  - Other items required by the definitions

Separate Files

- In C++ the ADT interface and implementation can be stored in separate files
  - The **interface file** stores the ADT interface
  - The **implementation file** stores the ADT implementation

A Minor Compromise

- The public part of the class definition is part of the ADT interface
- The private part of the class definition is part of the ADT implementation
  - This would hide it from those using the ADT
- C++ does not allow splitting the public and private parts of the class definition across files
  - The entire class definition is usually in the interface file

Case Study: DigitalTime

- The interface file of the DigitalTime ADT class contains the class 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 comments in the file should provide all the details needed to **use** the ADT

Naming The Interface File

- 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"
```

Display 9.1

#include " " or < > ?

- To include a predefined header file use < and >

  #include <iostream>

  - < and > tells the compiler to look where the system stores predefined header files
- To include a header file you wrote, use " and "

  #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)

#include "dtime.h"

- The implementation file requires an include directive to include the interface file:

#include "dtime.h"

Display 9.2 (1)

Display 9.2 (2)

Display 9.2 (3)

Display 9.2 (4)

The Application File

- The Application file is the file that contains the program that uses the ADT
  - It is also called a driver file
  - Must use an include directive to include the interface file:

```
#include "dtime.h"
```

Display 9.3

Running The Program

- Basic steps required to run a program:
  (Details vary from system to system!)
  - Compile the implementation file
  - Compile the application file
  - Link the files to create an executable program using a utility called a **linker**
    - Linking is often done automatically

Compile dtime.h ?

- The interface file is not compiled separately
  - The preprocessor replaces any occurrence of #include "dtime.h" with the text of dtime.h before compiling
  - 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

Why Three Files?

- Using separate files permits
  - The ADT to be used in other programs without rewriting the definition of the class for each
  - Implementation file to be compiled once even if multiple programs use the ADT
  - Changing the implementation file does not require changing the program using the ADT

Reusable Components

- An ADT coded in separate files can be used over and over
- The **reusability** of such an ADT class
  - Saves effort since it does not need to be
    - Redesigned
    - Recoded
    - Retested
  - Is likely to result in more reliable components

Multiple Classes

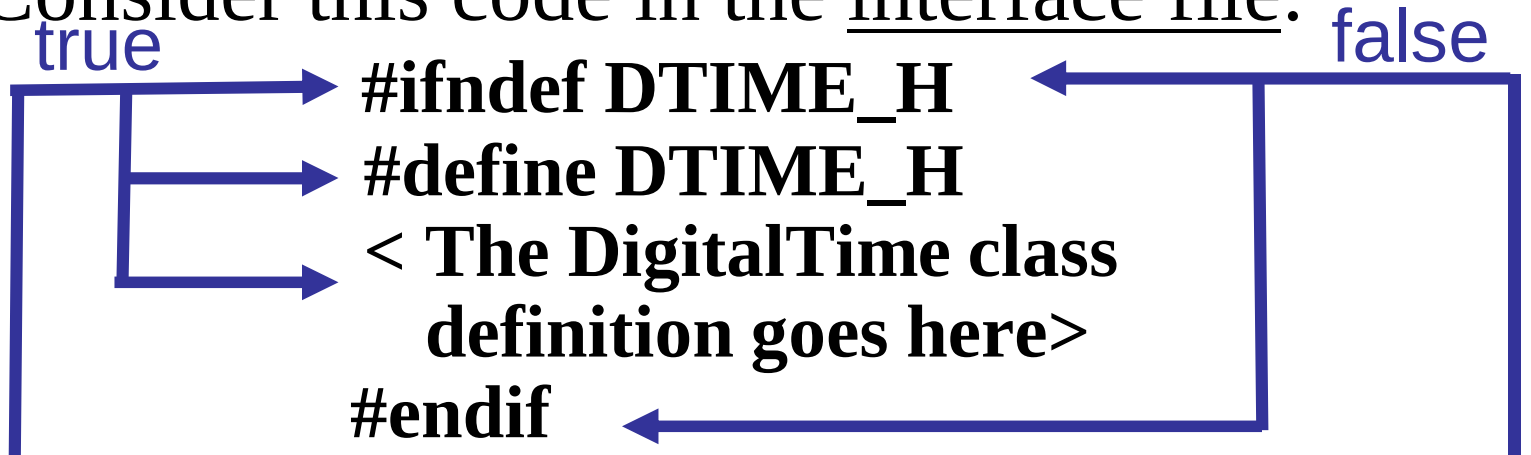
- A program may use several classes
  - Each could be stored in its own interface and implementation files
    - Some files can "include" other files, that include still others
  - It is possible that the same interface file could be included in multiple files
  - C++ does not allow multiple declarations 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 has been seen
  - **#ifndef DTIME_H**
    checks to see if DTIME_H has been defined
  - **#endif**
    If DTIME_H has been defined, skip to #endif

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

Why DTIME_H?

- 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 ' . '
- You may use any other identifier, but will make your code more difficult to read

Display 9.4

Defining Libraries

- You can create your own libraries of functions
  - You do not have to define a class to use separate files
  - If you have a collection of functions…
    - Declare them in a header file with their comments
    - Define them in an implementation file
    - Use the library files just as you use your class interface and implementation files

Section 9.1 Conclusion

- Can you
  - Determine which belongs to the interface, implementation or application files?
    - Class definition
    - Declaration of a non-member function used as an operation of the ADT
    - Definition of a member function
    - The main part of the program
  - Describe the difference between a C++ class and an ADT?

Namespaces

- A **namespace** is a collection of name definitions, such as class definitions and variable declarations
  - If a program uses classes and functions written by different programmers, it may be that the same name is used for different things
  - Namespaces help us deal with this problem

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)

The Global Namespace

- Code you write is in a namespace
  - it is in the **global namespace** unless you specify a namespace
  - The global namespace does not require the using directive

Name Conflicts

- If the same name is used in two namespaces
  - The namespaces cannot be used at the same time
  - Example: If my_function is defined in namespaces ns1 and ns2, the two versions of my_function could be used in one program by using local using directives this way:

```
{
  using namespace ns1;
  my_function( );
}
```

```
{
  using namespace ns2;
  my_function( );
}
```

Scope Rules For using

- A block is a list of statements enclosed in { }s
- The scope of a **using** directive is the block in which it appears
- A using directive placed at the beginning of a file, outside any block, applies to the entire file

Creating a Namespace

- To place code in a namespace
  - Use a **namespace grouping**
    - **namespace** ***Name_Space_Name***

      {

      Some_Code

      }
- To use the namespace created
  - Use the appropriate using directive
    - **using namespace** ***Name_Space_Name*****;**

Namespaces: Declaring a Function

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

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

Namespaces: Defining a Function

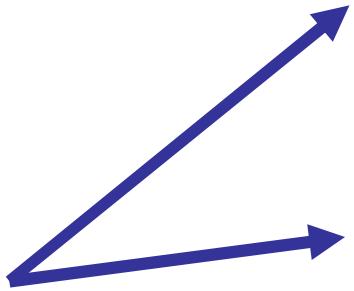
- To define a function declared in a namespace
  - Define the function in a namespace grouping

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

Namespaces: Using a Function

- 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

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



Using directive's scope

Display 9.5 (1-2)

A Namespace Problem

- Suppose you have the namespaces below:

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

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

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

Qualifying Names

- **Using declarations** (not directives) allow us to select individual functions to use from namespaces
  - **using ns1::fun1;** //makes **only** fun1 in ns1 available
    - 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( );

Qualifying Parameter Names

- To qualify the type of a parameter with a using declaration
  - Use the namespace and the type name

    int get_number (std::istream input_stream)

    …

    - istream is the istream defined in namespace std
    - If istream is the only name needed from namespace std, then you do not need to use

      using namespace std;

Directive/Declaration (Optional)

- A using declaration (using std::cout;) makes only one name available from the namespace
- A using directive makes all the names in the namespace available

A Subtle Point (Optional)

- A using directive potentially introduces a name
- If ns1 and ns2 both define my_function,

using namespace ns1;
using namespace ns2;

is OK, provided my_function is never used!

A Subtle Point Continued

- A using declaration introduces a name into your code: no other use of the name can be made

using ns1::my_function;
using ns2::my_function;

is illegal, even if my_function is never used

Unnamed Namespaces

- As we have done helper functions so far, they are not really hidden (Display 9.2)
  - We would like them to be local to the implementation file to implement information hiding
- The **unnamed namespace** can hide helper functions
  - Names defined in the unnamed namespace are local to the **compilation unit**
    - A compilation unit is a file (such as an implementation file) plus any file(s) #included in the file

The unnamed grouping

- Every compilation unit has an unnamed namespace
  - The namespace grouping is written as any other namespace, but no name is given:

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

Names In The unnamed namespace

- Names in the unnamed namespace
  - Can be reused outside the compilation unit
  - Can be used in the compilation unit without a namespace qualifier
- The rewritten version of the DigitalTime interface is found in **Display 9.6** while the implementation file is shown in **Display 9.7 (1)** **Display 9.7 (2)**

Namespaces In An Application

- The application file for the DigitalTime ADT is shown in **Display 9.8 (1)** **Display 9.8 (2)**

Compilation Units Overlap

- A header file is #included in two files
  - It is in two compilation units
  - Participates in two unnamed namespaces!
  - 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 the unnamed namespace of the implementation or application file

Naming 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

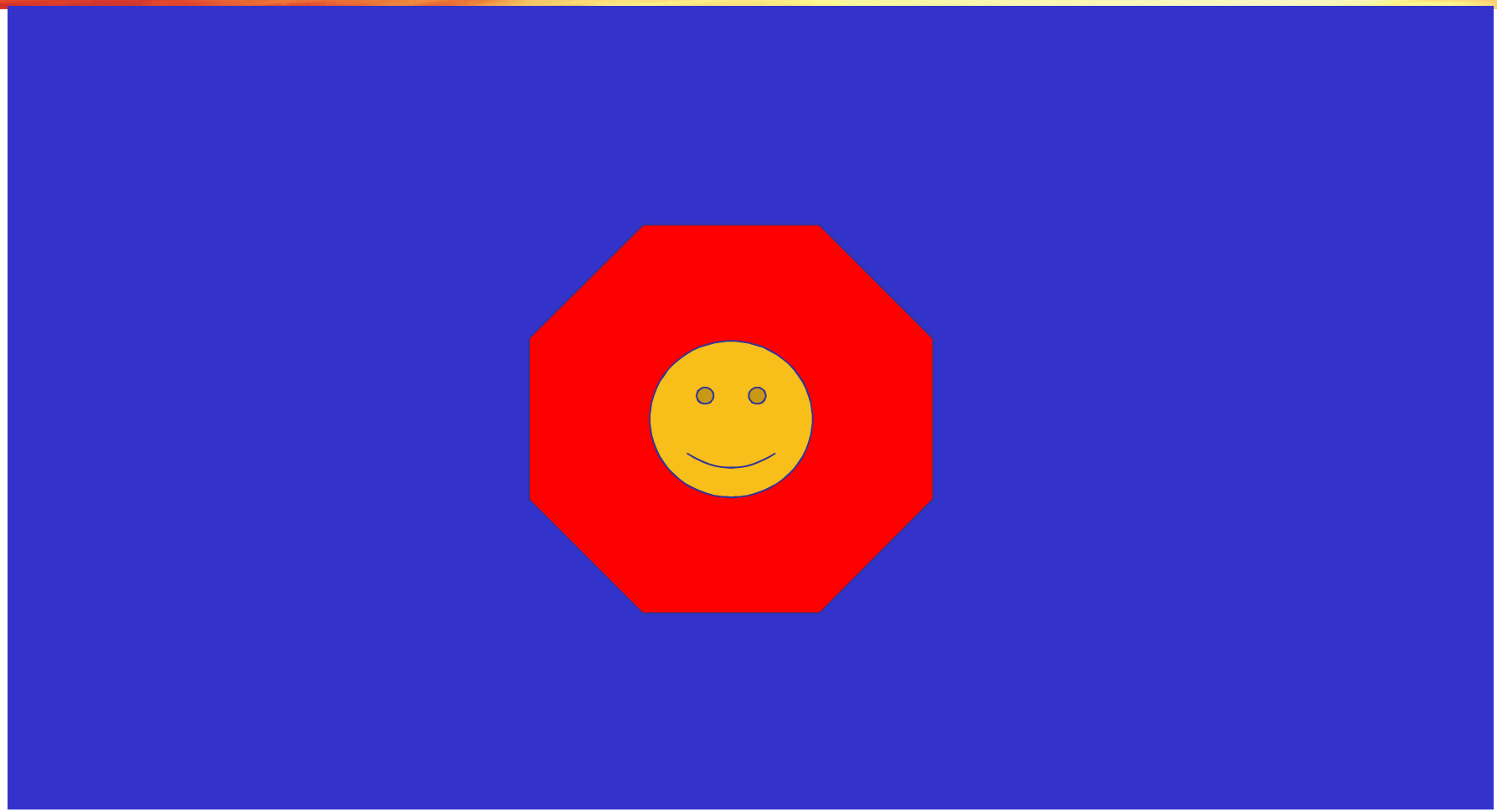
Global or unnamed?

- Names in the global namespace have global scope (all files)
  - They are available without a qualifier to all the program files
- Names in the unnamed namespace are local to a compilation unit
  - They are available without a qualifier within the compilation unit

Section 9.2 Conclusion

- Can you
  - Explain the purpose of using interface and implementation files?
  - Describe a namespace?
  - Demonstrate three ways to use the names in a namespace?

Chapter 9 -- End



Display 9.1

Interface File for `DigitalTime`

```
//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:
    friend bool operator ==(const DigitalTime& time1, const DigitalTime& time2);
    //Returns true if time1 and time2 represent the same time;
    //otherwise, returns false.

    DigitalTime(int the_hour, int the_minute);
    //Precondition: 0 <= the_hour <= 23 and 0 <= the_minute <= 59.
    //Initializes the time value to the_hour and the_minute.

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

    void advance(int minutes_added);
    //Precondition: The object has a time value.
    //Postcondition: The time has been changed to minutes_added minutes later.

    void advance(int hours_added, int minutes_added);
    //Precondition: The object has a time value.
    //Postcondition: The time value has been advanced
    //hours_added hours plus minutes_added minutes.

    friend istream& operator >>(istream& ins, DigitalTime& the_object);
    //Overloads the >> operator for input values of type DigitalTime.
    //Precondition: If ins is a file input stream, then ins has already been
    //connected to a file.

    friend ostream& operator <<(ostream& outs, const DigitalTime& the_object);
    //Overloads the << operator for output values of type DigitalTime.
    //Precondition: If outs is a file output stream, then outs has already been
    //connected to a file.
private:
    int hour;
    int minute;
};
```

For the definition of the types `istream` *and* `ostream`*, which are used as parameter types*

This is part of the implementation. It is not part of the interface. The word `private` *indicates that this is not part of the public interface.*

Display 9.2 (1/4)

Implementation File for `DigitalTime` (*part 1 of 4*)

```
//Implementation file dtime.cpp (Your system may require some
//suffix other than .cpp): This is the IMPLEMENTATION of the ADT DigitalTime.
//The interface for the class DigitalTime is in the header file dtime.h.
#include <iostream>
#include <cctype>
#include <cstdlib>
#include "dtime.h"
using namespace std;

//These FUNCTION DECLARATIONS are for use in the definition of
//the overloaded input operator >>:

void read_hour(istream& ins, int& the_hour);
//Precondition: Next input in the stream ins is a time in 24-hour notation,
//like 9:45 or 14:45.
//Postcondition: the_hour 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.

void read_minute(istream& ins, int& the_minute);
//Reads the minute from the stream ins after read_hour has read the hour.

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

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

//Uses iostream and cstdlib:
DigitalTime::DigitalTime(int the_hour, int the_minute)
{
    if (the_hour < 0 || the_hour > 23 || the_minute < 0 || the_minute > 59)
    {
        cout << "Illegal argument to DigitalTime constructor.";
        exit(1);
    }
```

Display 9.2 (2/4)

Implementation File for `DigitalTime` (*part 2 of 4*)

```
    else
    {
        hour = the_hour;
        minute = the_minute;
    }
}

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

void DigitalTime::advance(int minutes_added)
{
    int gross_minutes = minute + minutes_added;
    minute = gross_minutes%60;

    int hour_adjustment = gross_minutes/60;
    hour = (hour + hour_adjustment)%24;
}

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

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

Display 9.2 (3/4)

Implementation File for `DigitalTime` (*part 3 of 4*)

```
//Uses iostream:
istream& operator >>(istream& ins, DigitalTime& the_object)
{
    read_hour(ins, the_object.hour);
    read_minute(ins, the_object.minute);
    return ins;
}

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

//Uses iostream, cctype, and cstdlib:
void read_minute(istream& ins, int& the_minute)
{
    char c1, c2;
    ins >> c1 >> c2;

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

    the_minute = digit_to_int(c1)*10 + digit_to_int(c2);

    if (the_minute < 0 || the_minute > 59)
    {
        cout << "Error illegal input to read_minute\n";
        exit(1);
    }
}
```

Display 9.2 (4/4)

Implementation File for `DigitalTime` (*part 4 of 4*)

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

    if (isdigit(c1) && c2 == ':')
    {
        the_hour = digit_to_int(c1);
    }
    else //(isdigit(c1) && isdigit(c2))
    {
        the_hour = digit_to_int(c1)*10 + digit_to_int(c2);
        ins >> c2;//discard ':'
        if (c2 != ':')
        {
            cout << "Error illegal input to read_hour\n";
            exit(1);
        }
    }

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

Display 9.3

Application File Using `DigitalTime`

```
//Application file timedemo.cpp (your system may require some suffix
//other than .cpp): This program demonstrates use of the class DigitalTime.
#include <iostream>
#include "dtime.h"
using namespace std;

int main( )
{
    DigitalTime clock, old_clock;

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

    old_clock = clock;
    clock.advance(15);
    if (clock == old_clock)
        cout << "Something is wrong.";
    cout << "You entered " << old_clock << 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

```
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
```

Display 9.4

Avoiding Multiple Definitions of a Class

```
//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 namespace std;

class DigitalTime
{

<The definition of the class DigitalTime is the same as in Display 9.1.>


};

#endif //DTIME_H
```

Display 9.5 (1/2)

Namespace Demonstration (*part 1 of 2*)

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

namespace savitch1
{
    void greeting( );
}

namespace savitch2
{
    void greeting( );
}

void big_greeting( );

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

    {
        using namespace savitch1;
        greeting( );
    }

    big_greeting( );

    return 0;
}
```

Names in this block use definitions in namespaces `savitch2`, `std`, *and the global namespace.*

Names in this block use definitions in namespaces `savitch1`, `std`, *and the global namespace.*

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

Display 9.5 (2/2)

Namespace Demonstration (*part 2 of 2*)

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

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

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

Sample Dialogue

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

Display 9.6

Placing a Class in a Namespace—Header File

```
//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 namespace std;

namespace dtimesavitch
{

    class DigitalTime
    {

    <The definition of the class DigitalTime is the same as in Display 9.1.>
    };
}//end dtimesavitch

#endif //DTIME_H
```

One grouping for the namespace dtimesavitch. *Another grouping for the namespace* dtimesavitch *is in the implementation file* dtime.cpp.

Display 9.7 (1/2)

Placing a Class in a Namespace—Implementation File (*part 1 of 2*)

```
//Implementation file dtime.cpp (your system may require some
//suffix other than .cpp): This is the IMPLEMENTATION of the ADT DigitalTime.
//The interface for the class DigitalTime is in the header file dtime.h.
#include <iostream>
#include <cctype>
#include <cstdlib>
#include "dtime.h"
using namespace std;

namespace
{
    //These function declarations are for use in the definition of
    //the overloaded input operator >>:

    void read_hour(istream& ins, int& the_hour);
    //Precondition: Next input in the stream ins is a time in 24-hour notation,
    //like 9:45 or 14:45.
    //Postcondition: the_hour 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.

    void read_minute(istream& ins, int& the_minute);
    //Reads the minute from the stream ins after read_hour has read the hour.

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

namespace dtimesavitch
{
    bool operator ==(const DigitalTime& time1, const DigitalTime& time2)
    <The rest of the definition of == is the same as in Display 9.2.>

    DigitalTime::DigitalTime( )
    <The rest of the definition of this constructor is the same as in Display 9.2.>

    DigitalTime::DigitalTime(int the_hour, int the_minute)
    <The rest of the definition of this constructor is the same as in Display 9.2.>
```

One grouping for the unnamed namespace

One grouping for the namespace dtimesavitch. *Another grouping is in the file dtime.h.*

Display 9.7 (2/2)

Placing a Class in a Namespace—Implementation File (*part 2 of 2*)

```
    void DigitalTime::advance(int minutes_added)
    <The rest of the definition of this advance function is the same as in Display 9.2.>

    void DigitalTime::advance(int hours_added, int minutes_added)
    <The rest of the definition of this advance function is the same as in Display 9.2.>

    ostream& operator <<(ostream& outs, const DigitalTime& the_object)
    <The rest of the definition of << is the same as in Display 9.2.>

    //Uses iostream and functions in the unnamed namespace:
    istream& operator >>(istream& ins, DigitalTime& the_object)
    {
        read_hour(ins, the_object.hour);
        read_minute(ins, the_object.minute);
        return ins;
    }
} //dtimesavitch

namespace
{
    int digit_to_int(char c)
    <The rest of the definition of digit_to_int is the same as in Display 9.2.>

    void read_minute(istream& ins, int& the_minute)
    <The rest of the definition of read_minute is the same as in Display 9.2.>

    void read_hour(istream& ins, int& the_hour)
    <The rest of the definition of read_hour is the same as in Display 9.2.>

}//unnamed namespace
```

Functions defined in the unnamed namespace are local to this compilation unit (this file and included files). They can be used anywhere in this file, but have no meaning outside this compilation unit.

Another grouping for the unnamed namespace.

Display 9.8 (1/2)

Placing a Class 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 read_hour(int& the_hour);

int main( )
{
    using namespace std;

    using namespace dtimesavitch;

    int the_hour;
    read_hour(the_hour);

    DigitalTime clock(the_hour, 0), old_clock;

    old_clock = clock;
    clock.advance(15);
    if (clock == old_clock)
        cout << "Something is wrong.";
    cout << "You entered " << old_clock << 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;
}
```

If you place the using directives here, then the program behavior will be the same.

This is a different function `read_hour` *than the one in the implementation file* `dtime.cpp` *(shown in Display 9.7).*

Display 9.8 (2/2)

Placing a Class in a Namespace—Application Program (*part 2 of 2*)

```
void read_hour(int& the_hour)
{
    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 to 24): ";
    cin >> the_hour;
    if (the_hour == 24)
        the_hour = 0;
}
```