Separate Compilation and Namespaces 12

From mine own library with volumes that
I prize above my dukedom.

WILLIAM SHAKESPEARE, *The Tempest*

INTRODUCTION

This chapter covers two topics that have to do with how you organize a C++ program into separate parts. Section 12.1 on separate compilation discusses how a C++ program can be distributed across a number of files so that when some parts of the program change, only those parts need to be recompiled. The separate parts can also be more easily reused in other applications.

Section 12.2 discusses namespaces, which we introduced briefly in Chapter 2. Namespaces are a way of allowing you to reuse the names of classes, functions, and other items by qualifying the names to indicate different uses. Namespaces divide your code into sections so that the different sections may reuse the same names with differing meanings. Namespaces allow a kind of local meaning for names that is more general than local variables.

PREREQUISITES

This chapter uses material from Chapters 2 through 6 and 10 through 11.

12.1 SEPARATE COMPILATION

Your "if " is the only peacemaker; much virtue in "if."

WILLIAM SHAKESPEARE, *As You Like It*

C++ has facilities for dividing a program into parts that are kept in separate files, compiled separately, and then linked together when (or just before) the program is run. You can place the definition for a class (and its associated function definitions) in files that are separate from the programs that use the class. That way you can build up a library of classes so that many programs can use the same class. You can compile the class once and then use it in many different programs, just like you use the predefined libraries (such as those with header files `iostream` and `cstdlib`). Moreover, you can define the class itself in two files so that the specification of what the class does is separate from how the class is implemented. If your class is defined following the guidelines we have been giving you and you change only the implementation of the class, then you need only recompile the file with the class implementation. The other files, including the files with the programs that use the class, need not be changed or even recompiled. In this section, we tell you how to carry out this separate compilation of classes.

ADTs Reviewed

Recall that an ADT (abstract data type) is a class that has been defined so as to separate the interface and the implementation of the class. All your class definitions should be ADTs. In order to define a class so that it is an ADT, you need to separate the specification of how the class is used by a programmer from the details of how the class is implemented. The separation should be so complete that you can change the implementation without needing to change any program that uses the class in any way. The way to ensure this separation can be summarized in three rules:

1. Make all the member variables private members of the class.
2. Make each of the basic operations for the ADT (the class) either a public member function of the class, a friend function, an ordinary function, or an overloaded operator. Group the class definition and the function and operator declarations together. This group, along with its accompanying comments, is called the **interface** for the ADT. Fully specify how to use each such function or operator in a comment given with the class or with the function or operator declaration.
3. Make the implementation of the basic operations unavailable to the programmer who uses the abstract data type. The **implementation** consists of the function definitions and overloaded operator definitions (along with any helping functions or other additional items these definitions require).

In C++, the best way to ensure that you follow these rules is to place the interface and the implementation of the ADT class in separate files. As you might guess, the file that contains the interface is often called the **interface file,** and the file that contains the implementation is called the **implementation file.** The exact details of how to set up, compile, and use these files will vary slightly from one version of C++ to another, but the basic scheme is the same in all versions of C++. In particular, the details of what goes into the files are the same in all systems. The only things that vary are what commands you use to compile and link these files. The details about what goes into these files are illustrated in the next Case Study.

Private members are part of the implementation.

An ADT class has private member variables. Private member variables (and private member functions) present a problem to our basic philosophy of placing the interface and the implementation of an ADT in separate files. The public part of the class definition for an ADT is part of the interface for the ADT, but the private part is part of the implementation. This is a problem because C++ will not allow you to split the class definition across two files. Thus, some sort of compromise is needed. The only sensible compromise, and the one we use, is to place the entire class definition in the interface file. Since a programmer who is using the ADT class cannot use any of the private members of the class, the private members will, in effect, still be hidden from the programmer.

ADT

A data type is called an abstract data type (abbreviated ADT) if the programmers who use the type do not have access to the details of how the values and operations are implemented. All the classes that you define should be ADTs. An ADT class is a class that is defined following good programming practices that separate the interface and implementation of the class. (Any nonmember basic operations for the class such as overloaded operators are considered part of the ADT, even though they may not be officially part of the class definition.)

CASE STUDY *DigitalTime*—A Class Compiled Separately

Display 12.1 contains the interface file for an ADT class called `DigitalTime`. `DigitalTime` is a class whose values are times of day, such as 9:30. Only the public members of the class are part of the interface. The private members are part of the implementation, even though they are in the interface file. The label *`private`*: warns you that these private members are not part of the public interface. Everything that a programmer needs to know in order to use the ADT `DigitalTime` is explained in the comment at the start of the file and in the comments in the public section of the class definition. This interface tells the programmer how to use the two versions of the member function named `advance`, the constructors, and the overloaded operators `=`, `>>`, and `<<`. The member function named `advance`, the overloaded operators, and the assignment statement are the only ways that a programmer can manipulate objects and values of this class. As noted in the comment at the top of the interface file, this ADT class uses 24-hour notation, so, for instance, 1:30 PM is input and output as 13:30. This and the other details you must know in order to effectively use the class `DigitalTime` are included in the comments given with the member functions.

We have placed the interface in a file named `dtime.h`.The suffix .h indicates that this is a header file. An interface file is always a header file and therefore always ends with the suffix .h. Any program that uses the class `DigitalTime` must contain an include directive like the following, which names this file:

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

When you write an include directive, you must indicate whether the header file is a predefined header file that is provided for you or is a header file that you wrote. If the header file is predefined, write the header file name in angular brackets, like `<iostream>`. If the header file is one that you wrote, then write the header file name in quotes, like `"dtime.h"`. This distinction tells the compiler where to look for the header file. If the header file name is

DISPLAY 12.1 Interface File for `DigitalTime`

```
1   //Header file dtime.h: This is the INTERFACE for the class DigitalTime.
2   //Values of this type are times of day. The values are input and output in
3   //24-hour notation, as in 9:30 for 9:30 AM and 14:45 for 2:45 PM.
4   #include <iostream>
5   using namespace std;
6   class DigitalTime
7   {
8   public:
9      friend bool operator ==(const DigitalTime& time1, const DigitalTime& time2);
10       //Returns true if time1 and time2 represent the same time;
11       //otherwise, returns false.
12       DigitalTime(int the_hour, int the_minute);
13       //Precondition: 0 <= the_hour <= 23 and 0 <= the_minute <= 59.
14       //Initializes the time value to the_hour and the_minute.
15       DigitalTime( );
16       //Initializes the time value to 0:00 (which is midnight).
17       void advance(int minutes_added);
18       //Precondition: The object has a time value.
19       //Postcondition: The time has been changed to minutes_added minutes later.
20       void advance(int hours_added, int minutes_added);
21       //Precondition: The object has a time value.
22       //Postcondition: The time value has been advanced
23       //hours_added hours plus minutes_added minutes.
24       friend istream& operator >>(istream& ins, DigitalTime& the_object);
25       //Overloads the >> operator for input values of type DigitalTime.
26       //Precondition: If ins is a file input stream, then ins has already been
27       //connected to a file.
28       friend ostream& operator <<(ostream& outs, const DigitalTime& the_object);
29       //Overloads the << operator for output values of type DigitalTime.
30       //Precondition: If outs is a file output stream, then outs has already been
31       //connected to a file.
32  private:
33       int hour;
34       int minute;
35  };
```

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.*

in angular brackets, the compiler looks wherever the predefined header files are kept in your implementation of C++. If the header file name is in quotes, the compiler looks in the current directory or wherever programmer-defined header files are kept on your system.

Any program that uses our `DigitalTime` class must contain the previous include directive that names the header file `dtime.h`. That is enough to

allow you to compile the program but is not enough to allow you to run the program. In order to run the program, you must write (and compile) the definitions of the member functions and the overloaded operators. We have placed these function and operator definitions in another file, which is called the **implementation file.** Although it is not required by most compilers, it is traditional to give the interface file and the implementation file the same name. The two files do, however, end in different suffixes. We have placed the interface for our ADT class in the file named `dtime.h` and the implementation for our ADT class in a file named `dtime.cpp`. The suffix you use for the implementation file depends on your version of C++. Use the same suffix for the implementation file as you normally use for files that contain C++ programs. If your program files end in `.cxx`, then you would use `.cxx` in place of `.cpp`. If your program files end in `.CPP`, then your implementation files will end in `.CPP` instead of `.cpp`. We are using `.cpp` since most compilers accept `.cpp` as the suffix for a C++ source code file. The implementation file for our `DigitalTime` ADT class is given in Display 12.2. After we explain how the various files for our ADT interact with each other, we will return to Display 12.2 and discuss the details of the definitions in this implementation file.

In order to use the ADT class `DigitalTime` in a program, the program must contain the include directive

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

Notice that both the implementation file and the program file must contain this include directive that names the interface file. The file that contains the program (that is, the file that contains the main part of the program) is often called the **application file** or **driver file.** Display 12.3 contains an application file with a very simple program that uses and demonstrates the `DigitalTime` ADT class.

Compiling and running the program

The exact details on how you run this complete program, which is contained in three files, depend on what system you are using. However, the basic details are the same for all systems. You must compile the implementation file, and you must compile the application file that contains the main part of your program. You do not compile the interface file, which in this example is the file `dtime.h` given in Display 12.1. You do not need to compile the interface file because the compiler thinks the contents of this interface file are already contained in each of the other two files. Recall that both the implementation file and the application file contain the directive

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

Compiling your program automatically invokes a preprocessor that reads this include directive and replaces it with the text in the file `dtime.h`. Thus, the compiler sees the contents of `dtime.h`, and so the file `dtime.h` does not need to be compiled separately. (In fact, the compiler sees the contents of `dtime.h` twice: once when you compile the implementation file and once when you compile the application file.) This copying of the file `dtime.h` is

DISPLAY 12.2 Implementation File for `DigitalTime` *(part 1 of 3)*

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

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

11  void read_hour(istream& ins, int& the_hour);
12  //Precondition: Next input in the stream ins is a time in 24-hour notation,
13  //like 9:45 or 14:45.
14  //Postcondition: the_hour has been set to the hour part of the time.
15  //The colon has been discarded and the next input to be read is the minute.

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

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

21  bool operator ==(const DigitalTime& time1, const DigitalTime& time2)
22  {
23      return (time1.hour == time2.hour && time1.minute == time2.minute);
24  }

25  //Uses iostream and cstdlib:
26  DigitalTime::DigitalTime(int the_hour, int the_minute)
27  {
28      if (the_hour< 0 || the_hour> 23 || the_minute< 0 || the_minute> 59)
29      {
30          cout<< "Illegal argument to DigitalTime constructor.";
31          exit(1);
32      }
33
34      else
35      {
36          hour = the_hour;
37          minute = the_minute;
38      }
39  }
40  DigitalTime::DigitalTime( ) : hour(0), minute(0)
41  {
42      //Body intentionally empty.
43  }
44
```

(continued)

DISPLAY 12.2 Implementation File for `DigitalTime` *(part 2 of 3)*

```
45  void DigitalTime::advance(int minutes_added)
46  {
47      int gross_minutes = minute + minutes_added;
48      minute = gross_minutes % 60;
49
50      int hour_adjustment = gross_minutes / 60;
51      hour = (hour + hour_adjustment) % 24;
52  }
53
54  void DigitalTime::advance(int hours_added, int minutes_added)
55  {
56      hour = (hour + hours_added) % 24;
57      advance(minutes_added);
58  }
59
60  //Uses iostream:
61  ostream& operator <<(ostream& outs, const DigitalTime& the_object)
62  {
63      outs << the_object.hour<< ':';
64      if (the_object.minute< 10)
65          outs << '0';
66      outs << the_object.minute;
67      return outs;
68  }
69
70  //Uses iostream:
71  istream& operator >>(istream& ins, DigitalTime& the_object)
72  {
73      read_hour(ins, the_object.hour);
74      read_minute(ins, the_object.minute);
75      return ins;
76  }
77
78  int digit_to_int(char c)
79  {
80      return (static_cast <int>(c) - static_cast<int>('0'));
81  }
82
83  //Uses iostream, cctype, and cstdlib:
84  void read_minute(istream& ins, int& the_minute)
85  {
86      char c1, c2;
87      ins >> c1 >> c2;
88
89      if (!(isdigit(c1) && isdigit(c2)))
```

(continued)

DISPLAY 12.2 Implementation File for `DigitalTime` *(part 3 of 3)*

```
90          {
91              cout<< "Error illegal input to read_minute\n";
92              exit(1);
93          }
94
95          the_minute = (digit_to_int(c1) * 10) + digit_to_int(c2);
96
97          if (the_minute< 0 || the_minute> 59)
98          {
99              cout<< "Error illegal input to read_minute\n";
100             exit(1);
101         }
102     }
103
104     //Uses iostream, cctype, and cstdlib:
105     void read_hour(istream& ins, int& the_hour)
106     {
107         char c1, c2;
108         ins >> c1 >> c2;
109         if ( !( isdigit(c1) && (isdigit(c2) || c2 == ':' ) ) )
110         {
111             cout<< "Error illegal input to read_hour\n";
112             exit(1);
113         }
114
115         if (isdigit(c1) && c2 == ':')
116         {
117             the_hour = digit_to_int(c1);
118         }
119         else//(isdigit(c1) && isdigit(c2))
120         {
121             the_hour = (digit_to_int(c1) * 10) + digit_to_int(c2);
122             ins >> c2;//discard ':'
123             if (c2 != ':')
124             {
125                 cout<< "Error illegal input to read_hour\n";
126                 exit(1);
127             }
128         }
129         if (the_hour < 0 || the_hour > 23)
130         {
131             cout<< "Error illegal input to read_hour\n";
132             exit(1);
133         }
134     }
```

DISPLAY 12.3 Application File Using `DigitalTime`

```
1   //Application file timedemo.cpp (your system may require some suffix
2   //other than .cpp): This program demonstrates use of the class DigitalTime.
3   #include <iostream>
4   #include "dtime.h"
5   using namespace std;
6
7   int main( )
8   {
9       DigitalTime clock, old_clock;
10
11      cout<< "Enter the time in 24-hour notation: ";
12      cin>> clock;
13
14      old_clock = clock;
15      clock.advance(15);
16      if (clock == old_clock)
17         cout << "Something is wrong.";
18      cout << "You entered " << old_clock << endl;
19      cout << "15 minutes later the time will be "
20           << clock << endl;
21
22      clock.advance(2, 15);
23      cout << "2 hours and 15 minutes after that\n"
24           << "the time will be "
25           << clock << endl;
26
27      return 0;
28  }
```

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
```

only a conceptual copying. The compiler acts as if the contents of `dtime.h` were copied into each file that has the include directive. However, if you look in that file after it is compiled, you will only find the include directive; you will not find the contents of the file `dtime.h`.

Once the implementation file and the application file are compiled, you still need to connect these files so that they can work together. This is called

linking the files and is done by a separate utility called a **linker.** The details for how you call the linker depend on what system you are using. After the files are linked, you can run your program. (Often the linking is done automatically as part of the process of running the program.)

This process sounds complicated, but many systems have facilities that manage much of this detail for you automatically or semiautomatically. On any system, the details quickly become routine.

Why separate files?

Displays 12.1, 12.2, and 12.3 contain one complete program divided into pieces and placed in three different files. You could instead combine the contents of these three files into one file and then compile and run this one file without all this fuss about include directives and linking separate files. Why bother with three separate files? There are several advantages to dividing your program into separate files. Since you have the definition and the implementation of the class `DigitalTime` in files separate from the application file, you can use this class in many different programs without needing to rewrite the definition of the class in each of the programs. Moreover, you need to compile the implementation file only once, no matter how many programs use the class `DigitalTime`. But there are more advantages than that. Since you have separated the interface from the implementation of your `DigitalTime` ADT class, you can change the implementation file and will not need to change any program that uses the ADT. In fact, you will not even need to recompile the program. If you change the implementation file, you only need to recompile the implementation file and to relink the files. Saving a bit of recompiling time is nice, but the big advantage is not having to rewrite code. You can use the ADT class in many programs without writing the class code into each program. You can change the implementation of the ADT class and you need not rewrite any part of any program that uses the class.

Defining a Class in Separate Files: A Summary

You can define a class and place the definition of the class and the implementation of its member functions in separate files. You can then compile the class separately from any program that uses the class, and you can use this same class in any number of different programs. The class and the program that uses the class are placed in three files as follows:

1. Put the definition of the class in a header file called the **interface file.** The name of this header file ends in .h. The interface file also contains the declarations for any functions and overloaded operators that define basic operations for the class but that are not listed in the class definition. Include comments that explain how all these functions and operators are used.

(continued)

2. The definitions of all the functions and overloaded operators mentioned in step 1 (whether they are members or friends or neither) are placed in another file called the **implementation file.** This file must contain an include directive that names the interface file described above. This include directive uses quotes around the file name, as in the following example:

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

   The interface file and the implementation file traditionally have the same name, but end in different suffixes. The interface file ends in .h. The implementation file ends in the same suffix that you use for files that contain a complete C++ program. The implementation file is compiled separately before it is used in any program.

3. When you want to use the class in a program, place the main part of the program (and any additional function definitions, constant declarations, and so on) in another file called an **application file.** This file also must contain an include directive naming the interface file, as in the following example:

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

   The application file is compiled separately from the implementation file. You can write any number of these application files to use with one pair of interface and implementation files. To run an entire program, you must first link the object code that is produced by compiling the application file and the object code that is produced by compiling the implementation file. (On some systems the linking may be done automatically or semiautomatically.)

Implementation details

Now that we have explained how the various files in our ADT class and program are used, let's discuss the implementation of our ADT class (Display 12.2) in more detail. Most of the implementation details are straightforward, but there are two things that merit comment. Notice that the member function name `advance` is overloaded so that it has two function definitions. Also notice that the definition for the overloaded extraction (input) operator `>>` uses two "helping functions" called `read_hour` and `read_minute` and these two helping functions themselves use a third helping function called `digit_to_int`. Let's discuss these points.

The class `DigitalTime` (Displays 12.1 and 12.2) has two member functions called `advance`. One version takes a single argument, which is an integer giving the number of minutes to advance the time. The other version takes two arguments, one for a number of hours and one for a number of minutes, and advances the time by that number of hours plus that number of minutes. Notice that the definition of the two-argument version of `advance` includes a call to the one-argument version of `advance`. Look at the definition of the two-argument version that is given in Display 12.2. First the time is

advanced by hours_added hours, and then the single-argument version of advance is used to advance the time by an additional minutes_added minutes. At first this may seem strange, but it is perfectly legal. The two functions named advance are two different functions that, as far as the compiler is concerned, coincidentally happen to have the same name. The situation is no different in this regard than it would be if one of the two versions of the overloaded function advance had been called another_advance.

Now let's discuss the helping functions. The helping functions read_hour and read_minute read the input one character at a time and then convert the input to integer values that are placed in the member variables hour and minute. The functions read_hour and read_minute read the hour and minute one digit at a time, so they are reading values of type *char*. This is more complicated than reading the input as *int* values, but it allows us to perform error checking to see whether the input is correctly formed and to issue an error message if the input is not well formed. These helping functions read_hour and read_minute use another helping function named digit_to_int, which is the same as the digit_to_int function we used in our definition of the class Money in Displays 11.3. The function digit_to_int converts a digit, such as '3', to a number, such as 3.

Reusable Components

An ADT class developed and coded into separate files is a software component that can be used again and again in a number of different programs. **Reusability,** such as the reusability of these ADT classes, is an important goal to strive for when designing software components. A reusable component saves effort because it does not need to be redesigned, recoded, and retested for every application. A reusable component is also likely to be more reliable than a component that is used only once—for two reasons. First, you can afford to spend more time and effort on a component if it will be used many times. Second, if the component is used again and again, it is tested again and again. Every use of a software component is a test of that component. Using a software component many times in a variety of contexts is one of the best ways to discover any remaining bugs in the software.

Using #ifndef

We have given you a method for placing a program in three files: two for the interface and implementation of a class, and one for the application part of the program. A program can be kept in more than three files. For example, a program might use several classes, and each class might be kept in a separate pair of files.

VideoNote
Avoiding Multiple Definitions

Suppose you have a program spread across a number of files and more than one file has an include directive for a class interface file such as the following:

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

Under these circumstances, you can have files that include other files, and these other files may in turn include yet other files. This can easily lead to a situation in which a file, in effect, contains the definitions in `dtime.h` more than once. C++ does not allow you to define a class more than once, even if the repeated definitions are identical. Moreover, if you are using the same header file in many different projects, it becomes close to impossible to keep track of whether you included the class definition more than once. To avoid this problem, C++ provides a way of marking a section of code to say "if you have already included this stuff once before, do not include it again." The way this is done is quite intuitive, although the notation may look a bit weird until you get used to it. We will go through an example, explaining the details as we go.

The following directive "defines" `DTIME_H`:

```
#define DTIME_H
```

What this means is that the compiler's preprocessor puts `DTIME_H` on a list to indicate that `DTIME_H` has been seen. *Defines* is perhaps not the best word for this, since `DTIME_H` is not defined to mean anything but is merely put on a list. The important point is that you can use another directive to test whether or not `DTIME_H` has been defined and so test whether or not a section of code has already been processed. You can use any (nonkeyword) identifier in place of `DTIME_H`, but you will see that there are standard conventions for which identifier you should use.

The following directive tests to see whether or not `DTIME_H` has been defined:

```
#ifndef DTIME_H
```

If `DTIME_H` has already been defined, then everything between this directive and the first occurrence of the following directive is skipped:

```
#endif
```

(An equivalent way to state this, which may clarify the way the directives are spelled, is the following: If `DTIME_H` is *`not`* defined, then the compiler processes everything up to the next `#endif`. That *not* is why there is an `n` in `#ifndef`. This may lead you to wonder whether there is a `#ifdef` directive as well as a `#ifndef` directive. There is, and it has the obvious meaning, but we will have no occasion to use `#ifdef`.

Now consider the following code:

```
#ifndef DTIME_H
#define DTIME_H
<a class definition>
#endif
```

If this code is in a file named dtime.h, then no matter how many times your program contains

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

the class will be defined only one time.

The first time

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

is processed, the flag DTIME_H is defined and the class is defined. Now, suppose the compiler again encounters

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

When the include directive is processed this second time, the directive

```
#ifndef DTIME_H
```

says to skip everything up to

```
#endif
```

and so the class is not defined again.

In Display 12.4 we have rewritten the header file dtime.h shown in Display 12.1, but this time we used these directives to prevent multiple definitions. With the version of dtime.h shown in Display 12.4, if a file contains the following include directive more than once, the class DigitalTime will still be defined only once:

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

DISPLAY 12.4 Avoiding Multiple Definitions of a Class

```
1   //Header file dtime.h: This is the INTERFACE for the class DigitalTime.
2   //Values of this type are times of day. The values are input and output in
3   //24-hour notation, as in 9:30 for 9:30 AM and 14:45 for 2:45 PM.
4   #ifndef DTIME_H
5   #define DTIME_H
6   #include <iostream>
7   using namespace std;
8   class DigitalTime
9   {
10
    <The definition of the class DigitalTime is the same as in Display 12.1.>
11
12  };
13
14  #endif//DTIME_H
```

You may use some other identifier in place of `DTIME_H`, but the normal convention is to use the name of the file written in all uppercase letters with the underscore used in place of the period. You should follow this convention so that others can more easily read your code and so that you do not have to remember the flag name. This way the flag name is determined automatically and there is nothing arbitrary to remember.

These same directives can be used to skip over code in files other than header files, but we will not have occasion to use these directives except in header files.

PROGRAMMING TIP Defining Other Libraries

You need not define a class in order to use separate compilation. If you have a collection of related functions that you want to make into a library of your own design, you can place the function declarations and accompanying comments in a header file and the function definitions in an implementation file, just as we outlined for ADT classes. After that, you can use this library in your programs the same way you would use a class that you placed in separate files. ■

SELF-TEST EXERCISES

1. Suppose that you are defining an ADT class and that you then use this class in a program. You want to separate the class and program parts into separate files as described in this chapter. Specify whether each of the following should be placed in the interface file, implementation file, or application file:

   a. The class definition

   b. The declaration for a function that is to serve as an ADT operation, but that is neither a member nor a friend of the class

   c. The declaration for an overloaded operator that is to serve as an ADT operation, but that is neither a member nor a friend of the class

   d. The definition for a function that is to serve as an ADT operation, but that is neither a member nor a friend of the class

   e. The definition for a friend function that is to serve as an ADT operation

   f. The definition for a member function

   g. The definition for an overloaded operator that is to serve as an ADT operation, but that is neither a member nor a friend of the class

   h. The definition for an overloaded operator that is to serve as an ADT operation and that is a friend of the class

   i. The `main` part of your program

2. Which of the following files has a name that ends in `.h`: the interface file for a class, the implementation file for the class, or the application file that uses the class?

3. When you define a class in separate files, there is an interface file and an implementation file. Which of these files needs to be compiled? (Both? Neither? Only one? If so, which one?)

4. Suppose you define a class in separate files and use the class in a program. Now suppose you change the class implementation file. Which of the following files, if any, need to be recompiled: the interface file, the implementation file, or the application file?

5. Suppose you want to change the implementation of the class `DigitalTime` given in Displays 12.1 and 12.2. Specifically, you want to change the way the time is recorded. Instead of using the two private variables `hour` and `minute`, you want to use a single (private) `int` variable, which will be called `minutes`. In this new implementation, the private variable `minutes` will record the time as the number of minutes since the time 0:00 (that is, since midnight). So 1:30 is recorded as 90 minutes, since it is 90 minutes past midnight. Describe how you need to change the interface and implementation files shown in Displays 12.1 and 12.2. You need not write out the files in their entirety; just indicate what items you need to change and how, in a very general way, you would change them.

6. What is the difference between an ADT you define in C++ and a class you define in C++?

12.2 NAMESPACES

What's in a name? That which we call a rose
By any other name would smell as sweet.

WILLIAM SHAKESPEARE, *Romeo and Juliet*

When a program uses different classes and functions written by different programmers, there is a possibility that two programmers will use the same name for two different things. Namespaces are a way to deal with this problem. A namespace is a collection of name definitions, such as class definitions and variable declarations.

Namespaces and *using* Directives

We have already been using the namespace that is named `std`. The `std` namespace contains all the names defined in the standard library files (such

as iostream and cstdlib) that you use. For example, when you place the following at the start of a file,

```
#include <iostream>
```

that places all of the name definitions (for names like cin and cout) into the std namespace. Your program does not know about names in the std namespace unless you specify that it is using the std namespace. So far, the only way we know how to specify the std namespace (or any namespace) is with the following sort of *using* directive:

```
using namespace std;
```

A good way to see why you might want to include this *using* directive is to think about why you might want to *not* include it. If you do not include this *using* directive for the namespace std, then you can define cin and cout to have some meaning other than their standard meaning. (Perhaps you want to redefine cin and cout because you want them to behave a bit differently from the standard versions.) Their standard meaning is in the std namespace, and without the *using* directive (or something like it), your code knows nothing about the std namespace, and therefore, as far as your code is concerned, the only definitions of cin and cout are whatever definitions you give them.

Every bit of code you write is in some namespace. If you do not place the code in some specific namespace, then the code is in a namespace known as the **global namespace.** So far, we have not placed any code we wrote in any namespace, so all of our code has been in the global namespace. The global namespace does not have a *using* directive because you are always using the global namespace. You could say that there is always an implicit automatic *using* directive that says you are using the global namespace.

Note that you can be using more than one namespace at the same time. For example, we are always using the global namespace and we are usually using the std namespace. What happens if a name is defined in two namespaces and you are using both namespaces? This results in an error (either a compiler error or a run-time error, depending on the exact details). You can have the same name defined in two different namespaces, but if that is true, then you can only use one of those namespaces at a time.[1] However, this does not mean you cannot use the two namespaces in the same program. You can use them each at different times in the same program.

For example, suppose ns1 and ns2 are two namespaces, and suppose my_function is a *void* function with no arguments that is defined in both namespaces but defined in different ways in the two namespaces. The following is then legal:

---

[1]As you will see later in this chapter, there are ways to use two namespaces at the same time even if they contain the same name, but that is a subtle point that does not yet concern us.

```
{
    using namespace ns1;
    my_function( );
}
{
    using namespace ns2;
    my_function( );
}
```

The first invocation would use the definition of `my_function` given in the namespace `ns1`, and the second invocation would use the definition of `my_function` given in the namespace `ns2`.

Recall that a block is a list of statements, declarations, and possibly other code, enclosed in braces {}. A *using* directive at the start of a block applies only to that block. So the first *using* directive applies only in the first block, and the second *using* directive applies only in the second block. The usual way of phrasing this is to say that the **scope** of the `ns1` namespace is the first block, while the scope of the `ns2` namespace is the second block. Note that because of this scope rule, we are able to use two conflicting namespaces in the same program (such as in a program that contains the two blocks we discussed in the previous paragraph).

When you use a *using* directive in a block, it is typically the block consisting of the body of a function definition. If you place a *using* directive at the start of a file (as we have usually done so far), then the *using* directive applies to the entire file. A *using* directive should normally be placed near the start of a file or the start of a block.

Scope Rule for *using* Directives

The scope of a *using* directive is the block in which it appears (more precisely, from the location of the *using* directives to the end of the block). If the *using* directive is outside of all blocks, then it applies to all of the file that follows the *using* directive.

Creating a Namespace

In order to place some code in a namespace, you simply place it in a namespace grouping of the following form:

```
namespace Name_Space_Name
{
    Some_Code
}
```

When you include one of these groupings in your code, you are said to place the names defined in *Some_Code* into the namespace *Name_Space_Name*. These

names (really the definitions of these names) can be made available with the *using* directive:

```
using namespace Name_Space_Name;
```

For example, the following, taken from Display 12.5, places a function declaration in the namespace savitch1:

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

If you look again at Display 12.5, you see that the definition of the function greeting is also placed in namespace savitch1. That is done with the following additional namespace grouping:

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

Note that you can have any number of these namespace groupings for a single namespace. In Display 12.5, we used two namespace groupings for namespace savitch1 and two other groupings for namespace savitch2.

Every name defined in a namespace is available inside the namespace grouping, but the names can be also be made available to code outside of the namespace. That function declaration and function definition in the namespace savitch1 can be made available with the *using* directive:

```
using namespace savitch1
```

as illustrated in Display 12.5.

DISPLAY 12.5 Namespace **Demonstration** *(part 1 of 2)*

```
 1   #include <iostream>
 2   using namespace std;
 3
 4   namespace savitch1
 5   {
 6       void greeting( );
 7   }
 8
 9   namespace savitch2
10   {
11       void greeting( );
12   }
```

(continued)

DISPLAY 12.5 Namespace Demonstration *(part 2 of 2)*

```
13
14  void big_greeting( );
15
16  int main( )
17  {
18      {
19          using namespace savitch2;
20          greeting( );
21      }
22
23      {
24          using namespace savitch1;
25          greeting( );
26      }
27
28      big_greeting( );
29
30      return 0;
31  }
32
33  namespace savitch1
34  {
35      void greeting( )
36      {
37          cout << "Hello from namespace savitch1.\n";
38      }
39  }
40
41  namespace savitch2
42  {
43      void greeting( )
44      {
45          cout<< "Greetings from namespace savitch2.\n";
46      }
47  }
48
49  void big_greeting( )
50  {
51      cout<< "A Big Global Hello!\n";
52  }
```

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

Sample Dialogue

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

SELF-TEST EXERCISES

7. Consider the program shown in Display 12.5. Could we use the name `greeting` in place of `big_greeting`?

8. In Self-Test Exercise 7, we saw that you could *not* add a definition for the following function (to the global namespace):

```
void greeting( );
```

Can you add a definition for the following function declaration to the global namespace?

```
void greeting(int how_many);
```

9. Can a namespace have more than one namespace grouping?

Qualifying Names

Suppose you are faced with the following situation: You have two namespaces, `ns1` and `ns2`. You want to use the function `fun1` defined in `ns1` and the function `fun2` defined in namespace `ns2`. The complication is that both `ns1` and `ns2` define a function `my_function`. (Assume all functions in this discussion take no arguments, so overloading does not apply.) It would not be a good idea to use the following:

```
using namespace ns1;
using namespace ns2;
```

This would provide conflicting definitions for `my_function`.

What you need is a way to say you are using `fun1` in namespace `ns1` and `fun2` in namespace `ns2` and nothing else in the namespaces `ns1` and `ns2`. The following are called ***using* declarations,** and they are your answer:

```
using ns1::fun1;
using ns2::fun2;
```

A *using* declaration of the form

```
using Name_Space::One_Name
```

makes (the definition of) the name *One_Name* from the namespace *Name_Space* available, but does not make any other names in *Name_Space* available.

Note that you have seen the scope resolution operator, `::`, before. For example, in Display 12.2 we had the following function definition:

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

In this case the :: means that we are defining the function advance for the class DigitalTime, as opposed to any other function named advance in any other class. Similarly,

```
using ns1::fun1;
```

means we are using the function named fun1 as defined in the namespace ns1, as opposed to any other definition of fun1 in any other namespace.

Now suppose that you intend to use the name fun1 as defined in the namespace ns1, but you intend to use it only one time (or a small number of times). You can then name the function (or other item) using the name of the namespace and the scope resolution operator as in the following:

```
ns1::fun1( );
```

This form is often used when specifying a parameter type. For example, consider

```
int get_number(std::istream input_stream)
. . .
```

In the function get_number, the parameter input_stream is of type istream, where istream is defined as in the std namespace. If this use of the type name istream is the only name you need from the std namespace (or if all the names you need are similarly qualified with std::), then you do *not* need

```
using namespace std;
```

A Subtle Point About Namespaces (*Optional*)

There are two differences between a *using* declaration, such as

```
using std::cout;
```

and a *using* directive, such as

```
using namespace std;
```

The differences are as follows:

1. A *using* declaration (like *using* std::cout;) makes only one name in the namespace available to your code, while a *using* directive (like *using namespace* std;) makes all the names in a namespace available.
2. A *using* declaration introduces a name (like cout) into your code so that no other use of the name can be made. However, a *using* directive only potentially introduces the names in the namespace.

Point 1 is pretty obvious. Point 2 has some subtleties. For example, suppose the namespaces ns1 and ns2 both provide definitions for my_function but have no other name conflicts. Then the following will produce no problems:

```
using namespace ns1;
using namespace ns2;
```

provided that (within the scope of these directives) the conflicting name `my_function` is never used in your code. On the other hand, the following is illegal, even if the function `my_function` is never used:

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

Sometimes this subtle point can be important, but it does not impinge on most routine code.

SELF-TEST EXERCISES

10. Write the function declaration for a *void* function named `wow`. The function `wow` has two parameters, the first of type `speed` as defined in the `speedway` namespace and the second of type `speed` as defined in the `indy500` namespace.

11. Consider the following function declarations from the definition of the class `Money` in Display 11.4.

    ```
    void input(istream& ins);
    void output(ostream& outs) const;
    ```

    Rewrite these function declarations so that they do not need to be preceded by

    ```
    using namespace std;
    ```

    (You do not need to look back at Display 11.4 to do this.)

Unnamed Namespaces

Our definition of the class `DigitalTime` in Displays 12.1 and 12.2 used three helping functions: `digit_to_int`, `read_hour`, and `read_minute`. These helping functions are part of the implementation for the ADT class `DigitalTime`, so we placed their definitions in the implementation file (Display 12.2). However, this does not really hide these three functions. We would like these functions to be local to the implementation file for the class `DigitalTime`. However, as we have done it, they are not. In particular, we cannot define another function with the name `digit_to_int` (or `read_hour` or `read_minute`) in an application program that uses the class `DigitalTime`. This violates the principle of information hiding. To truly hide these helping functions and make them local to the implementation file for `DigitalTime`, we need to place them in a special namespace called the *unnamed namespace*.

A **compilation unit** is a file, such as a class implementation file, along with all the files that are `#included` in the file, such as the interface header file for the class. Every compilation unit has an **unnamed namespace.** A namespace grouping for the unnamed namespace is written in the same way as any other namespace, but no name is given, as in the following example:

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

All the names defined in the unnamed namespace are local to the compilation unit, and thus the names can be reused for something else outside the compilation unit. For example, Displays 12.6 and 12.7 show a rewritten (and our final) version of the interface and implementation file for the class `DigitalTime`. Note that the helping functions (`read_hour`, `read_minute`, and `digit_to_int`) are all in the unnamed namespace and therefore are local to the compilation unit. As illustrated in Display 12.8, the names in the unnamed namespace can be reused for something else outside the compilation unit. In Display 12.8 the function name `read_hour` is reused for another different function in the application program.

DISPLAY 12.6 Placing a Class in a Namespace—Header File

```
1   //Header file dtime.h: This is the interface for the class DigitalTime.
2   //Values of this type are times of day. The values are input and output in
3   //24-hour notation, as in 9:30 for 9:30 AM and 14:45 for 2:45 PM.
4
5   #ifndef DTIME_H
6   #define DTIME_H
7
8   #include <iostream>
9   using namespace std;
10
11  namespace dtimesavitch
12  {
13
14      class DigitalTime
15      {
16
      <The definition of the class DigitalTime is the same as in Display 12.1.>
17      };
18  }//end dtimesavitch
19
20  #endif //DTIME_H
```

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

DISPLAY 12.7 Placing a Class in a Namespace—Implementation File *(part 1 of 2)*

```
1   //Implementation file dtime.cpp (your system may require some
2   //suffix other than .cpp): This is the IMPLEMENTATION of the ADT DigitalTime.
3   //The interface for the class DigitalTime is in the header file dtime.h.
4   #include <iostream>
5   #include <cctype>
6   #include <cstdlib>
7   #include "dtime.h"
8   using namespace std;
9
10  namespace
11  {
12      //These function declarations are for use in the definition of
13      //the overloaded input operator >>:
14
15      void read_hour(istream& ins, int& the_hour);
16      //Precondition: Next input in the stream ins is a time in 24-hour notation,
17      //like 9:45 or 14:45.
18      //Postcondition: the_hour has been set to the hour part of the time.
19      //The colon has been discarded and the next input to be read is the minute.
20
21      void read_minute(istream& ins, int& the_minute);
22      //Reads the minute from the stream ins after read_hour has read the hour.
23
24      int digit_to_int(char c);
25      //Precondition: c is one of the digits '0' through '9'.
26      //Returns the integer for the digit; for example, digit_to_int('3')
27      //returns 3.
28  }//unnamed namespace
29
30
31  namespace dtimesavitch
32  {
33      bool operator ==(const DigitalTime& time1, const DigitalTime& time2)
        <The rest of the definition of == is the same as in Display 12.2.>
34
35      DigitalTime::DigitalTime( )
        <The rest of the definition of this constructor is the same as in Display 12.2.>
36
37      DigitalTime::DigitalTime(int the_hour, int the_minute)
        <The rest of the definition of this constructor is the same as in Display 12.2.>
38      void DigitalTime::advance(int minutes_added)
        <The rest of the definition of this advance function is the same as in Display 12.2.>
39
40      void DigitalTime::advance(int hours_added, int minutes_added)
        <The rest of the definition of this advance function is the same as in Display 12.2.>
41
```

One grouping for the unnamed namespace

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

(continued)

DISPLAY 12.7 Placing a Class in a Namespace—Implementation File *(part 2 of 2)*

```
42        ostream& operator <<(ostream& outs, const DigitalTime& the_object)
        <The rest of the definition of << is the same as in Display 12.2.>
43
44        //Uses iostream and functions in the unnamed namespace:
45        istream& operator >>(istream& ins, DigitalTime& the_object)
46        {
47            read_hour(ins, the_object.hour);
48            read_minute(ins, the_object.minute);
49            return ins;
50        }
51    } //dtimesavitch
52
53
54    namespace
55    {
56        int digit_to_int(char c)
        <The rest of the definition of digit_to_int is the same as in Display 12.2.>
57
58        void read_minute(istream& ins, int& the_minute)
        <The rest of the definition of read_minute is the same as in Display 12.2.>
59
60        void read_hour(istream& ins, int& the_hour)
        <The rest of the definition of read_hour is the same as in Display 12.2.>
61
62    }//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.

If you look again at the implementation file in Display 12.8, you will see that the helping functions `digit_to_int`, `read_hour`, and `read_minute` are used outside the unnamed namespace without any namespace qualifier. Any name defined in the unnamed namespace can be used without qualification anywhere in the compilation unit. (Of course, this needed to be so, since the unnamed namespace has no name to use for qualifying its names.)

It is interesting to note how unnamed namespaces interact with the C++ rule that you cannot have two definitions of a name (in the same namespace). There is one unnamed namespace in each compilation unit. It is easily possible for compilation units to overlap. For example, both the implementation file for a class and an application program using the class would normally include the header file (interface file) for the class. Thus, the header file is in two compilation units and hence participates in two unnamed namespaces. As dangerous as this sounds, it will normally produce no problems as long as each compilation unit's namespace makes sense when considered by itself. For example, if a name is defined in the unnamed namespace in the header file, it cannot be defined again in the unnamed namespace in either the implementation file or the application file. So, a name conflict is avoided.

DISPLAY 12.8 Placing a Class in a Namespace—Application Program *(part 1 of 2)*

```
1   //This is the application file: timedemo.cpp. This program
2   //demonstrates hiding the helping functions in an unnamed namespace.
3
4   #include <iostream>
5   #include "dtime.h"
6
7   void read_hour(int& the_hour);
8
9   int main( )
10  {
11      using namespace std;
12
13      using namespace dtimesavitch;
14
15      int the_hour;
16      read_hour(the_hour);
17
18      DigitalTime clock(the_hour, 0), old_clock;
19
20      old_clock = clock;
21      clock.advance(15);
22      if (clock == old_clock)
23          cout << "Something is wrong.";
24      cout << "You entered " << old_clock << endl;
25      cout << "15 minutes later, the time will be "
26           << clock; << endl;
27
28      clock.advance(2, 15);
29      cout << "2 hours and 15 minutes after that\n"
30           << "the time will be "
31           << clock; << endl;
32
33      return 0;
34  }
35  void read_hour(int& the_hour)
36  {
37      using namespace std;
38
39      cout << "Let's play a time game.\n"
40           << "Let's pretend the hour has just changed.\n"
41           << "You may write midnight as either 0 or 24,\n"
42           << "but I will always write it as 0.\n"
43           << "Enter the hour as a number (0 to 24): ";
44      cin >> the_hour;
45      if (the_hour == 24)
46           the_hour = 0;
47  }
```

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 12.7).*

(continued)

DISPLAY 12.8 Placing a Class in a Namespace—Application Program *(part 2 of 2)*

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
```

PROGRAMMING TIP Choosing a Name for a Namespace

It is a good idea to include your last name or some other unique string in the names of your namespaces so as to reduce the chance that somebody else will use the same namespace name as you do. With multiple programmers writing code for the same project, it is important that namespaces that are meant to be distinct really do have distinct names. Otherwise, you can easily have multiple definitions of the same names in the same scope. That is why we included the name `savitch` in the namespace `dtimesavitch` in Display 12.7. ■

Unnamed Namespace

You can use the **unnamed namespace** to make a definition local to a compilation unit (that is, to a file and its included files). Each compilation unit has one unnamed namespace. All the identifiers defined in the unnamed namespace are local to the compilation unit. You place a definition in the unnamed namespace by placing it in a namespace grouping with no namespace name, as shown in the following:

```
namespace
{
    Definition_1
    Definition_2
        .
        .
        .
    Definition_Last
}
```

You can use any name in the unnamed namespace without a qualifier anyplace in the compilation unit. See Displays 12.6 and 12.7 for a complete example.

PITFALL Confusing the Global Namespace and the Unnamed Namespace

Do not confuse the global namespace with the unnamed namespace. If you do not put a name definition in a namespace grouping, then it is in the global namespace. To put a name definition in the unnamed namespace, you must put it in a namespace grouping that starts as follows, without a name:

```
namespace
{
```

Both names in the global namespace and names in the unnamed namespace may be accessed without a qualifier. However, names in the global namespace have global scope (all the program files), while names in an unnamed namespace are local to a compilation unit.

This confusion between the global namespace and the unnamed namespace does not arise very much in writing code, since there is a tendency to think of names in the global namespace as being "in no namespace," even though that is not technically correct. However, the confusion can easily arise when discussing code. ■

SELF-TEST EXERCISES

12. Would the program in Display 12.8 behave any differently if you replaced the *using* directive

```
using namespace dtimesavitch;
```

with the following *using* declaration?

```
using dtimesavitch::DigitalTime;
```

13. What is the output produced by the following program?

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

namespace sally
{
    void message( );
}

namespace
{
    void message( );
}

int main( )
{
    {
        message( );
```

```
        using sally::message;
        message( );
    }

    message( );
    return 0;
}
namespace sally
{
    void message( )
    {
        cout << "Hello from Sally.\n";
    }
}
namespace
{
    void message( )
    {
        cout << "Hello from unnamed.\n";
    }
}
```

14. In Display 12.7 there are two groupings for the unnamed namespace: one for the helping function declarations and one for the helping function definitions. Can we eliminate the grouping for the helping function declarations? If so, how can we do it?

CHAPTER SUMMARY

- In C++, abstract data types (ADTs) are implemented as classes with all member variables private and with the operations implemented as public member and nonmember functions and overloaded operators.
- You can define an ADT as a class and place the definition of the class and the implementation of its member functions in separate files. You can then compile the ADT class separately from any program that uses it and you can use this same ADT class in any number of different programs.
- A namespace is a collection of name definitions, such as class definitions and variable declarations.
- There are three ways to use a name from a namespace: by making all the names in the namespace available with a *using* directive, by making the single name available by a *using* declaration for the one name, or by qualifying the name with the name of the namespace and the scope resolution operator.
- You place a definition in a namespace by placing it in a namespace grouping for that namespace.
- The unnamed namespace can be used to make a name definition local to a compilation unit.

Answers to Self-Test Exercises

1. Parts (a), (b), and (c) go in the interface file; parts (d) through (h) go in the implementation file. (All the definitions of ADT operations of any sort go in the implementation file.) Part (i) (that is, the `main` part of your program) goes in the application file.

2. The name of the interface file ends in `.h`.

3. Only the implementation file needs to be compiled. The interface file does not need to be compiled.

4. Only the implementation file needs to be recompiled. You do, however, need to relink the files.

5. You need to delete the private member variables `hour` and `minute` from the interface file shown in Display 12.1 and replace them with the member variable `minutes` (with an `s`). You do not need to make any other changes in the interface file. In the implementation file, you need to change the definitions of all the constructors and other member functions, as well as the definitions of the overloaded operators, so that they work for this new way of recording time. (In this case, you do not need to change any of the helping functions `read_hour`, `read_minute`, or `digit_to_int`, but that might not be true for some other class or even some other reimplementation of this class.) For example, the definition of the overloaded operator `>>` could be changed to the following:

```
istream& operator >>(istream& ins, DigitalTime& the_object)
{
    int input_hour, input_minute;
    read_hour(ins, input_hour);
    read_minute(ins, input_minute);
    the_object.minutes = input_minute + (60 * input_hour);
    return ins;
}
```

   You need not change any application files for programs that use the class. However, since the interface file is changed (as well as the implementation file), you will need to recompile any application files, and of course you will need to recompile the implementation file.

6. The short answer is that an ADT is simply a class that you defined following good programming practices of separating the interface from the implementation. Also, when we describe a class as an ADT, we consider the nonmember basic operations such as overloaded operators to be part of the ADT, even though they are not technically speaking part of the C++ class.

7. No. If you replace `big_greeting` with `greeting`, then you will have a definition for the name `greeting` in the global namespace. There are parts of the program where all the name definitions in the namespace `savitch1` and all the name definitions in the global namespace are simultaneously available. In those parts of the program, there would be two distinct definitions for

   ```
   void greeting( );
   ```

8. Yes, the additional definition would cause no problems. This is because overloading is always allowed. When, for example, the namespaces `savitch1` and the global namespace are available, the function name `greeting` would be overloaded. The problem in Self-Test Exercise 7 was that there would sometimes be two definitions of the function name `greeting` with the same parameter lists.

9. Yes, a namespace can have any number of groupings. For example, the following are two groupings for the namespace `savitch1` that appear in Display 12.5:

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

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

10. ```
    void wow(speedway::speed s1, indy500::speed s2);
    ```

11. ```
    void input(std::istream& ins);
    ```

    ```
    void output(std::ostream& outs) const;
    ```

12. The program would behave exactly the same.

13. `Hello from unnamed. Hello from Sally. Hello from unnamed.`

14. Yes, you can eliminate the grouping for the helping function declarations, as long as the grouping with the helping function definitions occurs before the helping functions are used. For example, you could remove the namespace with the helping function declarations and move the grouping with the helping function definitions to just before the namespace grouping for the namespace `dtimesavitch`.

PRACTICE PROGRAMS

Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.

1. Add the following member function to the ADT class `DigitalTime` defined in Displays 12.1 and 12.2:

```
void DigitalTime::interval_since(const DigitalTime& a_previous_time,
    int& hours_in_interval, int& minutes_in_interval) const
```

This function computes the time interval between two values of type `DigitalTime`. One of the values of type `DigitalTime` is the object that calls the member function `interval_since`, and the other value of type `DigitalTime` is given as the first argument. For example, consider the following code:

```
DigitalTime current(5, 45), previous(2, 30);
int hours, minutes;
current.interval_since(previous, hours, minutes);
cout << "The time interval between " << previous
     << " and " << current << endl
     << "is " << hours << " hours and "
<< minutes << " minutes.\n";
```

In a program that uses your revised version of the `DigitalTime` ADT, this code should produce the following output:

```
The time interval between 2:30 and 5:45
is 3 hours and 15 minutes.
```

Allow the time given by the first argument to be later in the day than the time of the calling object. In this case, the time given as the first argument is assumed to be on the previous day. You should also write a program to test this revised ADT class.

2. Do Self-Test Exercise 5 in full detail. Write out the complete ADT class, including interface and implementation files. Also write a program to test your ADT class.

VideoNote
Solution to Practice Program 12.3

3. Redo Practice Programs 1 from Chapter 11, but this time define the `Money` ADT class in separate files for the interface and implementation so that the implementation can be compiled separately from any application program.

4. Redo Practice Programs 2 from Chapter 11, but this time define the `Pairs` ADT class in separate files for the interface and implementation so that the implementation can be compiled separately from any application program.

5. This Practice Program explores how the unnamed namespace works. Listed below are snippets from a program to perform input validation for a username and password. The code to input and validate the username is in a file separate from the code to input and validate the password.

   File `user.cpp`:

```
namespace Authenticate
{
    void inputUserName()
    {
        do
        {
        cout << "Enter your username (8 letters only)"  << endl;
        cin >> username;
        } while (!isValid());
    }
    string getUserName()
    {
        return username;
    }
}
```

   Define the `username` variable and the `isValid()` function in the unnamed namespace so the code will compile. The `isValid()` function should return `true` if username contains exactly eight letters. Generate an appropriate header file for this code.

   Repeat the same steps for the file `password.cpp`, placing the `password` variable and the `isValid()` function in the unnamed namespace. In this case, the `isValid()` function should return `true` if the input password has at least eight characters including at least one nonletter:

   File `password.cpp`:

```
namespace Authenticate
{
    void inputPassword()
    {
        do
        {
        cout << "Enter your password (at least 8 characters " <<
                "and at least one nonletter)"  << endl;
        cin >> password;
        } while (!isValid());
    }
    string getPassword()
    {
        return password;
    }
}
```

At this point you should have two functions named `isValid()`, each in different unnamed namespaces. Place the following `main` function in an appropriate place. The program should compile and run.

```
int main()
{
    inputUserName();
    inputPassword();
    cout << "Your username is " << getUserName() <<
            " and your password is: " <<
            getPassword() << endl;
    return 0;
}
```

Test the program with several invalid usernames and passwords.

PROGRAMMING PROJECTS

Programming Projects require more problem-solving than Practice Programs and can usually be solved many different ways. Visit www.myprogramminglab.com to complete many of these Programming Projects online and get instant feedback.

1. Redo (or do for the first time) Practice Program 3 from Chapter 11. Define your ADT class in separate files so that it can be compiled separately.

2. Redo (or do for the first time) Programming Project 2 from Chapter 11. Define your ADT class in separate files so that it can be compiled separately.

3. Redo (or do for the first time) Programming Project 9 from Chapter 11. Define your ADT class in separate files so that it can be compiled separately. Put the main function in its own file separate from the ADT files.