EE1003
Introduction to Computer I

Chapter 9
Classes

Andy, Yu-Guang Chen
Assistant Professor, Department of EE
National Central University
andyygchen@ee.ncu.edu.tw

Learning Objectives

In this chapter you'll learn:

- How to define a class and use it to create an object.
- To implement a class's behaviors as member functions.
- How to implement a class's attributes as data members.
- How to use a constructor to initialize an object's data when the object is created.
- How to separate a class's interface and implementation.
- To prevent multiple definition errors with preprocessor wrappers.
- To understand class scope and accessing class members.
- How destructors are used to perform "termination housekeeping" on an object.
- When constructors and destructors are called.
- The logic errors that may occur when a `public` member function returns a reference to `private` data.

Outline

9.1 Introduction

- In this chapter, you'll begin writing programs that employ the basic concepts of object-oriented programming.
- Typically, programs consist of function `main` and one or more classes, each containing data members and member functions.
- In this chapter, we develop a simple, well-engineered framework for organizing object-oriented programs.

9.2 Classes, Objects, Member Functions and Data Members

- Let's begin with a simple analogy to help you reinforce your understanding of classes and their contents.
- Suppose you want to drive a car and make it go faster by pressing down on its accelerator pedal.
- What must happen before you can do this? Well, before you can drive a car, someone has to *design it and build it.*
- In a sense, the pedal "hides" the complex mechanisms that actually make the car go faster.
- People with little or no knowledge of how cars are engineered can drive a car easily, simply by using the user-friendly "interfaces" to the car's complex internal mechanisms.
  - Accelerator pedal, brake pedal, steering wheel, transmission shifting …

9.2 Classes, Objects, Member Functions and Data Members

- Unfortunately, you cannot drive the engineering drawings of a car—so before you can drive a car, it must be built from the engineering drawings that describe it.
- A completed car will have an *actual* accelerator pedal to make the car go faster.
- But even that's not enough—the car will not accelerate on its own, so the driver must press the accelerator pedal to tell the car to go faster.

9.2 Classes, Objects, Member Functions and Data Members

- Now let's use our car example to introduce the key object-oriented programming concepts of this section.
- Performing a task in a program requires a function.
- In C++, we create a program unit called a *class* to house a function
  - A function belonging to a class is called a member function.
- The function describes the actually performed operations.
  - In a class, you provide one or more member functions that are designed to perform the class's tasks.
- The function hides from its user the complex tasks.
  - Just as the accelerator pedal of a car hides from the driver the complex mechanisms of making the car go faster.

9.2 Classes, Objects, Member Functions and Data Members

- Just as you cannot drive an engineering drawing of a car, you cannot "drive" a class.
- You must create an object of a class before you can get a program to perform the tasks the class describes.
  - Just as someone has to build a car from its engineering drawings before you can actually drive the car.
- Many cars can be built from the same engineering drawing. → Many objects can be built from the same class.
  - That's why C++ is called an object-oriented programming language.
- When you drive a car, pressing its gas pedal sends a message to the car to make the car go faster.
- Similarly, you send messages to an object and tells a member function of the object to perform its task.

9.2 Classes, Objects, Member Functions and Data Members

- In addition to the capabilities a car provides, it also has many attributes.
  - Such as its color, the number of doors, the amount of gas in its tank, its current speed and its total miles driven (i.e., its odometer reading).
- Like the car's capabilities, these attributes are represented as part of a car's design in its engineering diagrams.
- As you drive a car, these attributes are always associated with the car.
- Every car maintains its own attributes.
- For example, each car knows how much gas is in its own gas tank, but not how much is in the tanks of other cars.

9.2 Classes, Objects, Member Functions and Data Members

- Similarly, an object has attributes that are carried with it as it's used in a program.
- These attributes are specified as part of the object's class.
- For example, a bank account object has a balance attribute that represents the amount of money in the account.
- Each bank account object knows the balance in the account it represents, but not the balances of the other accounts in the bank.
- Attributes are specified by the class's data members.

9.3 Time Class

- We begin with an example (Fig. 9.1) that consists of class `Time` (lines 8–19), which represents the time of day in 24-hour clock format, the class's member functions (lines 23–50) and a `main` function (lines 52–79) that creates and manipulates a `Time` object.
- Function `main` uses this object and its member functions to set and display the time in both 24-hour and 12-hour formats.

9.3 Time Class

```
1  // Fig. 9.1: fig09_01.cpp
2  // Time class.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  // Time class definition
8  class Time
9  {
10 public:
11    Time(); // constructor
12    void setTime( int, int, int ); // set hour, minute and second
13    void printUniversal(); // print time in universal-time format
14    void printStandard(); // print time in standard-time format
15 private:
16    int hour; // 0 - 23 (24-hour clock format)
17    int minute; // 0 - 59
18    int second; // 0 - 59
19 }; // end class Time
20
```

Fig. 9.1 | `Time` class definition. (Part 1 of 4.)

9.3 Time Class

```
// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time()
{
   hour = minute = second = 0;
} // end Time constructor

// set new Time value using universal time; ensure that
// the data remains consistent by setting invalid values to zero
void Time::setTime( int h, int m, int s )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
   minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
   second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
} // end function setTime

// print Time in universal-time format (HH:MM:SS)
void Time::printUniversal()
{
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
      << setw( 2 ) << minute << ":" << setw( 2 ) << second;
} // end function printUniversal

```

Fig. 9.1 | Time class definition. (Part 2 of 4.)

9.3 Time Class

```
// print Time in standard-time format (HH:MM:SS AM or PM)
void Time::printStandard()
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":"
      << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 )
      << second << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard

int main()
{
   Time t; // instantiate object t of class Time

   // output Time object t's initial values
   cout << "The initial universal time is ";
   t.printUniversal(); // 00:00:00
   cout << "\nThe initial standard time is ";
   t.printStandard(); // 12:00:00 AM

   t.setTime( 13, 27, 6 ); // change time

   // output Time object t's new values
   cout << "\n\nUniversal time after setTime is ";
   t.printUniversal(); // 13:27:06
```

Fig. 9.1 | Time class definition. (Part 3 of 4.)

9.3 Time Class

```
67      cout << "\nStandard time after setTime is ";
68      t.printStandard(); // 1:27:06 PM
69
70      t.setTime( 99, 99, 99 ); // attempt invalid settings
71
72      // output t's values after specifying invalid values
73      cout << "\n\nAfter attempting invalid settings:"
74         << "\nUniversal time: ";
75      t.printUniversal(); // 00:00:00
76      cout << "\nStandard time: ";
77      t.printStandard(); // 12:00:00 AM
78      cout << endl;
79   } // end main
```

```
The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Fig. 9.1 | Time class definition. (Part 4 of 4.)

9.3 Time Class

- Before creating a `Time` object, we must tell the compiler what member functions and data members belong to the class
  - A process known as defining the class.
- The `Time` class definition (lines 8–19) begins with keyword `class` followed by the class name `Time` (line 8).
- By convention, the name of a class begins with a capital letter
  - Each subsequent word in the class name begins with a capital letter.
- This capitalization style is often referred to as camel case.
- Every class's body is enclosed in a pair of left and right braces (`{` and `}`), as in lines 8 and 19.
- The class definition terminates with a semicolon (line 19).

Common Programming Error 9.1
Forgetting the semicolon at the end of a class definition is a syntax error.

9.3 Time Class

- Line 10 contains the access-specifier label public:.
  - Access specifiers are always followed by a colon (`:`).
- These functions appear after `public:` are the `public` member functions of the class
  - Also known as the interface of the class.
- We provide four public member functions in class `Time` — `Time`, `setTime`, `printUniversal` and `printStandard`.
  - These services allow the client code to interact with an object of the class to manipulate the class's data..
- We'll soon see that classes can have non-`public` member functions as well.

9.3 Time Class

- Lines 16–18 declare three integer members to represent the hour, minute and second, respectively.
  - These declarations appear after the access-specifier label private:.
- Variables or functions declared after access specifier `private` (and before the next access specifier) are accessible only to member functions of the class for which they're declared.
  - Cannot be accessed by functions outside the class (such as `main`).
- Normally, data members are listed in the `private` portion of a class and member functions are listed in the `public` portion.
  - Declaring data members with access specifier `private` is known as data hiding.
- It's possible to have `private` member functions and `public` data, as we'll see later.

9.3 Time Class

Software Engineering Observation 9.1

Generally, data members should be declared `private` *and member functions should be declared* `public`*. It's appropriate to declare certain member functions* `private`*, if they're to be accessed only by other member functions of the class.*

Good Programming Practice 9.1

For clarity and readability, use each access specifier only once in a class definition. Place `public` *members first, where they're easy to locate.*

9.3 Time Class

Common Programming Error 9.2

An attempt by a function, which is not a member of a particular class (or a `friend` *of that class, as we'll see in Chapter 10, Classes: A Deeper Look), to access a* `private` *member of that class is a compilation error.*

Software Engineering Observation 9.2

Each element of a class should be `private` *unless it can be proven that the element needs to be* `public`*. This is another example of the principle of least privilege.*

9.3 Time Class

- The member function with the same name as the class is called a constructor.
- This is a special member function that initializes the data members of a class object.
- A class's constructor is called when a program creates an object of that class.
- If a class does not explicitly include a constructor, the compiler provides a default constructor
  - A constructor with no parameters and no actions.
- It's common to have several constructors for a class, enabling objects to be initialized several ways.
- Constructors cannot specify a return type.

9.3 Time Class

- The `Time` constructor (lines 23–26) initializes the data members to 0—the universal-time equivalent of 12 AM.
  - Called when the `Time` object is created to ensure that the object begins in a consistent state.
- Invalid values cannot be stored in the data members of a `Time` object, because all subsequent attempts by a client to modify the data members are scrutinized by function `setTime`.
- It's strongly recommended that these data members be initialized by the class's constructor
  - Private data members cannot be initialized directly.
- Data members can also be assigned values by `Time`'s other member functions.

9.3 Time Class

- Function `setTime` (lines 30–35) is a `public` function that declares three `int` parameters and uses them to set the time.
- A conditional expression tests each argument to determine whether the value is in a specified range.
- In class `Time`, invalid values are set to zero to ensure that the object's data values are always kept in range.
  - Even if the provided arguments were incorrect.
- A value passed to `setTime` is a correct value if it's in the allowed range for the member it's initializing.
  - Ex: any number in the range 0–23 would be a correct value for the `hour`.

9.3 Time Class

- Function `printUniversal` (lines 38–42) takes no arguments and outputs the time in universal-time format (e.g., 13:27:06).
- Parameterized stream manipulator setfill specifies the fill character that appear to the left of the digits in the number.
  - If the number being output fills the specified field, the fill character will not be displayed.
- Once the fill character is specified with `setfill`, it applies for all subsequent values (i.e., `setfill` is a "sticky" setting).
  - This is in contrast to `setw`, which applies only to the next value.
- Function `printStandard` (lines 45–50) takes no arguments and outputs the date in standard-time format (e.g., 1:27:06 PM).
- `setfill('0')` is used to format the `minute` and `second` as two digit values with leading zeros if necessary.

9.3 Time Class

Error-Prevention Tip 9.1

Each sticky setting (such as a fill character or floating-point precision) should be restored to its previous setting when it's no longer needed. Failure to do so may result in incorrectly formatted output later in a program. Chapter 15, Stream Input/Output, discusses how to reset the fill character and precision.

9.3 Time Class

- Each member-function name in the function headers (lines 23, 30, 38 and 45) is preceded by the class name and `::`, which is known as the binary scope resolution operator.
- This "ties" each member function to the `Time` class definition.
  - After "tied" to the class, it is still within that class's scope.
- Without "`Time::`" preceding each function name, these functions would not be recognized by the compiler as member functions of class `Time`
  - The compiler would consider them as global functions, like `main`.
  - Such functions cannot access class `Time`'s `private` data or call its member functions, without specifying an object.

9.3 Time Class

Common Programming Error 9.3

When defining a class's member functions outside that class, omitting the class name and binary scope resolution operator (::) preceding the function names causes compilation errors.

9.3 Time Class

- Typically, you cannot call a member function of a class until you create an object of that class.
- Line 54 creates an object of class `Time` called `t`.
  - The variable's type is `Time`.
- The compiler does not automatically know what type `Time` is—it's a user-defined type.
  - We tell the compiler what `Time` is by including the class definition.
- Each class you create becomes a new type that can be used to create objects.
  - Can be used in object, array, pointer and reference declarations
- When the object is instantiated (line 54), the `Time` constructor is called to initialize each `private` data member to `0`.

9.3 Time Class

- Lines 58 and 60 print the time in universal and standard formats to confirm the values of data members.
- These two member-function calls each use variable `t` followed by the dot operator (`.`), the function name and an empty set of parentheses.
- At the beginning of line 58, "`t.`" indicates that `main` should use the `Time` object that was created in line 54.
- The empty parentheses indicate that member function `printUniversal` does not require additional data to perform its task.

9.3 Time Class

- Note that the data members `hour`, `minute` and `second` (lines 16–18) are preceded by the `private` member access specifier.
  - `private` data members are not accessible outside the class.
- The philosophy here is that the data representation used within the class is of no concern to the class's clients.
  - For example, the class can represent the time internally as the number of seconds since midnight.
  - Clients could use the same `public` member functions and get the same results without being aware of this.
- The implementation of a class is said "*hidden from its clients*".
- Classes simplify programming because the user of the class object need only be concerned with the operations encapsulated in the object.

9.3 Time Class

- The `printUniversal` and `printStandard` member functions take no arguments, because these member functions implicitly know that they're to print the data members of the particular `Time` object for which they're invoked.
- This can make member function calls more concise than conventional function calls in procedural programming.

9.3 Time Class

Software Engineering Observation 9.3

Using an object-oriented programming approach often simplifies function calls by reducing the number of parameters. This benefit of object-oriented programming derives from the fact that encapsulating data members and member functions within an object gives the member functions the right to access the data members.

9.3 Time Class

Software Engineering Observation 9.4

Member functions are usually shorter than functions in non-object-oriented programs, because the data stored in data members have ideally been validated by a constructor or by member functions that store new data. Because the data is already in the object, the member-function calls often have no arguments or fewer arguments than typical function calls in non-object-oriented languages. Thus, the calls are shorter, the function definitions are shorter and the function prototypes are shorter. This improves many aspects of program development.

9.3 Time Class

Error-Prevention Tip 9.2

The fact that member function calls generally take either no arguments or substantially fewer arguments than conventional function calls in non-object-oriented languages reduces the likelihood of passing the wrong arguments, the wrong types of arguments or the wrong number of arguments.

9.3 Time Class

- Often, classes do not have to be created "from scratch." Rather, they can include objects of other classes as members or they may be derived from other classes that provide attributes and behaviors the new classes can use.
- Such software reuse can greatly enhance productivity and simplify code maintenance.
- Including class objects as members of other classes is called composition (or aggregation) and is discussed in Chapter 10.
- Deriving new classes from existing classes is called inheritance and is discussed in Chapter 12.

9.3 Time Class

- People new to object-oriented programming often suppose that objects must be quite large because they contain data members and member functions.
- Logically, this is true—you may think of objects as containing data and functions (and our discussion has certainly encouraged this view); physically, however, this is not true.

9.3 Time Class

Performance Tip 9.1

Objects contain only data, so objects are much smaller than if they also contained member functions. Applying operator `sizeof` *to a class name or to an object of that class will report only the size of the class's data members. The compiler creates one copy (only) of the member functions separate from all objects of the class. All objects of the class share this one copy. Each object, of course, needs its own copy of the class's data, because the data can vary among the objects. The function code is nonmodifiable and, hence, can be shared among all objects of one class.*

9.4 Class Scope and Accessing Class Members

- A class's data members (variables declared in the class definition) and member functions (functions declared in the class definition) belong to that class's scope.
- Within a class's scope, class members are immediately accessible by all of that class's member functions and can be referenced by name.
- Outside a class's scope, `public` class members are referenced through one of the handles on an object
  - An object name, a reference to an object or a pointer to an object.
- The type of the object, reference or pointer specifies the interface (i.e., the member functions) accessible to the client.

9.4 Class Scope and Accessing Class Members

- Member functions of a class can be overloaded, but only by other member functions of that class.
- To overload a member function, simply provide in the class definition a prototype for each version of the overloaded function, and provide a separate function definition for each version of the function.
- Variables declared in a member function have local scope and are known only to that function.
- If a member function defines a variable with the same name as a variable with class scope, the class-scope variable is hidden by the block-scope variable in the local scope.
- Such a hidden variable can be accessed by preceding the variable name with the class name followed by the scope resolution operator (`::`).

9.4 Class Scope and Accessing Class Members

- The dot member selection operator (`.`) is preceded by an object's name or with a reference to an object to access the object's members.
- The arrow member selection operator (->) is preceded by a pointer to an object to access the object's members.
- Figure 9.2 uses a simple class called `Count` (lines 7–24) with `private` data member `x` of type `int` (line 23), `public` member function `setX` (lines 11–14) and `public` member function `print` (lines 17–20) to illustrate accessing class members with the member-selection operators.

9.4 Class Scope and Accessing Class Members

```cpp
// Fig. 9.2: fig09_02.cpp
// Demonstrating the class member access operators . and ->
#include <iostream>
using namespace std;

// class Count definition
class Count
{
public: // public data is dangerous
   // sets the value of private data member x
   void setX( int value )
   {
      x = value;
   } // end function setX

   // prints the value of private data member x
   void print()
   {
      cout << x << endl;
   } // end function print

```

Fig. 9.2 | Accessing an object's member functions through each type of object handle—the object's name, a reference to the object and a pointer to the object. (Part 1 of 3.)

9.4 Class Scope and Accessing Class Members

```cpp
private:
   int x;
}; // end class Count

int main()
{
   Count counter; // create counter object
   Count *counterPtr = &counter; // create pointer to counter
   Count &counterRef = counter; // create reference to counter

   cout << "Set x to 1 and print using the object's name: ";
   counter.setX( 1 ); // set data member x to 1
   counter.print(); // call member function print

   cout << "Set x to 2 and print using a reference to an object: ";
   counterRef.setX( 2 ); // set data member x to 2
   counterRef.print(); // call member function print

   cout << "Set x to 3 and print using a pointer to an object: ";
   counterPtr->setX( 3 ); // set data member x to 3
   counterPtr->print(); // call member function print
} // end main
```

```
Set x to 1 and print using the object's name: 1
Set x to 2 and print using a reference to an object: 2
Set x to 3 and print using a pointer to an object: 3
```

9.4 Class Scope and Accessing Class Members

- The member functions in Fig. 9.2 are defined completely in the body of class `Count`'s definition.
- In such cases, the compiler attempts to inline calls to the member function.
  - This can improve performance.
- Only the simplest and most stable member functions (i.e., whose implementations are unlikely to change) are defined in the class header.

Software Engineering Observation 9.5

Defining a small member function inside the class definition does not promote the best software engineering, because clients of the class will be able to see the implementation of the function, and the client code must be recompiled if the function definition changes.

9.5 Placing a Class in a Separate File for Reusability

- One of the benefits of creating class definitions is that, when packaged properly, our classes can be reused by programmers—potentially worldwide.
- For example, we can reuse C++ Standard Library type `string` in any C++ program by including the header file `<string>`.
- Programmers who wish to use our `Time` class cannot simply include the file from Fig. 9.1 in another program.
- If other programmers include the code from Fig. 9.1, they get extra baggage—our `main` function—and their programs will then have two `main` functions.
  - Every program must have exactly one `main` function.

9.5 Placing a Class in a Separate File for Reusability

- The previous examples in the chapter consist of a single `.cpp` file, also known as a source-code file, that contains a class definition and a `main` function.
- When building an object-oriented C++ program, it's customary to define reusable source code (such as a class) in a file that by convention has a `.h` filename extension—known as a header file.
- Programs use `#include` preprocessor directives to include header files and take advantage of reusable software components, such as type `string` provided in the C++ Standard Library and user-defined types like class `Time`.

9.6 Time Class: Separating Interface from Implementation

- We now show how to promote software reusability by separating a class definition from the client code (e.g., function `main`) that uses the class.
- We also introduce another fundamental principle of good software engineering—separating interface from implementation.

9.6 Time Class: Separating Interface from Implementation

- Interfaces define and standardize the ways in which things such as people and systems interact with one another.
- The interface specifies *what* operations a radio permits users to perform but does not specify *how* the operations are implemented inside the radio.
- The interface of a class describes what services a class's clients can use and how to request those services, but not how the class carries out the services.
- A class's `public` interface consists of the class's `public` member functions (also known as the class's public services).

9.6 Time Class: Separating Interface from Implementation

- In our prior examples, each class definition contained the complete definitions of the class's `public` member functions and the declarations of its `private` data members.
- It's better software engineering to define member functions *outside* the class definition, so that their implementation details can be hidden from the client code.
  - This practice ensures that you do not write client code that depends on the class's implementation details.
  - If you were to do so, the client code would be more likely to "break" if the class's implementation changed.

9.6 Time Class: Separating Interface from Implementation

- This example separates class `Time`'s interface from its implementation, and the class definition from the client code
- The definition is split into two files—the header file `Time.h` (Fig. 9.3) and the source-code file `Time.cpp` (Fig. 9.4).
  - The header file defines class `Time`.
  - The source-code file defines `Time`'s member functions.
- By convention, member-function definitions are placed in a source-code file of the same base name (e.g., `Time`) as the class's header file but with a `.cpp` filename extension.
- The source-code file `fig09_05.cpp` (Fig. 9.5) defines function `main` (the client code).
- The code and output of Fig. 9.5 are identical to that of Fig. 9.1.

9.6 Time Class: Separating Interface from Implementation

- Header file `Time.h` (Fig. 9.3) contains the `Time`'s class definition (lines 10–21).
- Again, the compiler must know the data members of the class to determine how much memory to reserve for each object of the class.
- Including the header file `Time.h` in the client code (line 5 of Fig. 9.5) provides the compiler with the information it needs
  - Ensure that the appropriate amount of memory is reserved for each `Time` object.
  - Ensure that client code calls class `Time`'s member functions correctly.

9.6 Time Class: Separating Interface from Implementation

```cpp
1  // Fig. 9.3: Time.h
2  // Declaration of class Time.
3  // Member functions are defined in Time.cpp
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME_H
7  #define TIME_H
8
9  // Time class definition
10 class Time
11 {
12 public:
13    Time(); // constructor
14    void setTime( int, int, int ); // set hour, minute and second
15    void printUniversal(); // print time in universal-time format
16    void printStandard(); // print time in standard-time format
17 private:
18    int hour; // 0 - 23 (24-hour clock format)
19    int minute; // 0 - 59
20    int second; // 0 - 59
21 }; // end class Time
22
23 #endif
```

Fig. 9.3 | Time class definition.

9.6 Time Class: Separating Interface from Implementation

- In Fig. 9.3, the class definition is enclosed in the following preprocessor wrapper (lines 6, 7 and 23):

```cpp
#ifndef TIME_H
#define TIME_H
    ...
#endif
```

- If the header has not been included previously, the name `TIME_H` is defined by the `#define` directive and the following statements are included.
- If the header has been included previously, `TIME_H` is defined already and the header file is not included again.
  - In large programs with many header files, some header files may have been included in another header files.

9.6 Time Class: Separating Interface from Implementation

- Source-code file `Time.cpp` (Fig. 9.4) defines class `Time`'s member functions, which are identical to the member-function definitions of Fig. 9.1.
- To indicate that the member functions in `Time.cpp` are part of class `Time`, we must first include the `Time.h` header file (line 5).
- When compiling `Time.cpp`, the compiler uses the information in `Time.h` to ensure that
  - the first line of each member function (lines 10, 17, 25 and 32) matches its prototype in the `Time.h`
  - each member function knows about the class's data members and other member.

9.6 Time Class: Separating Interface from Implementation

```
1  // Fig. 9.4: Time.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4  #include <iomanip>
5  #include "Time.h" // include definition of class Time from Time.h
6  using namespace std;
7
8  // Time constructor initializes each data member to zero.
9  // Ensures all Time objects start in a consistent state.
10 Time::Time()
11 {
12    hour = minute = second = 0;
13 } // end Time constructor
14
15 // set new Time value using universal time; ensure that
16 // the data remains consistent by setting invalid values to zero
17 void Time::setTime( int h, int m, int s )
18 {
19    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
20    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
21    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
22 } // end function setTime
23
```

Fig. 9.4 | Time class member-function definitions. (Part 1 of 2.)

9.6 Time Class: Separating Interface from Implementation

```cpp
// print Time in universal-time format (HH:MM:SS)
void Time::printUniversal()
{
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
      << setw( 2 ) << minute << ":" << setw( 2 ) << second;
} // end function printUniversal

// print Time in standard-time format (HH:MM:SS AM or PM)
void Time::printStandard()
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":"
      << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 )
      << second << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
```

Fig. 9.4 | Time class member-function definitions. (Part 2 of 2.)

9.6 Time Class: Separating Interface from Implementation

- To help you prepare for the larger programs you'll encounter going forward in this book and in industry, we often use a separate source-code file containing function `main` to test our classes (this is called a driver program).
- Figure 9.5 performs the same `Time` object manipulations as Fig. 9.1.
- Separating `Time`'s interface from its member-function implementation does not affect the way that this client code uses the class.
- It affects only how the program is compiled and linked, which we discuss in detail shortly.

9.6 Time Class: Separating Interface from Implementation

Software Engineering Observation 9.8

Information important to the interface of a class should be included in the header file. Information that will be used only internally in the class and will not be needed by clients of the class should be included in the unpublished source file. This is yet another example of the principle of least privilege.

9.6 Time Class: Separating Interface from Implementation

```
// Fig. 9.5: fig09_05.cpp
// Program to test class Time.
// NOTE: This file must be compiled with Time.cpp.
#include <iostream>
#include "Time.h" // include definition of class Time from Time.h
using namespace std;

int main()
{
   Time t; // instantiate object t of class Time

   // output Time object t's initial values
   cout << "The initial universal time is ";
   t.printUniversal(); // 00:00:00
   cout << "\nThe initial standard time is ";
   t.printStandard(); // 12:00:00 AM

   t.setTime( 13, 27, 6 ); // change time

   // output Time object t's new values
   cout << "\n\nUniversal time after setTime is ";
   t.printUniversal(); // 13:27:06
   cout << "\nStandard time after setTime is ";
   t.printStandard(); // 1:27:06 PM
```

Fig. 9.5 | Program to test class `Time`. (Part 1 of 2.)

9.6 Time Class: Separating Interface from Implementation

```
25
26      t.setTime( 99, 99, 99 ); // attempt invalid settings
27
28      // output t's values after specifying invalid values
29      cout << "\n\nAfter attempting invalid settings:"
30         << "\nUniversal time: ";
31      t.printUniversal(); // 00:00:00
32      cout << "\nStandard time: ";
33      t.printStandard(); // 12:00:00 AM
34      cout << endl;
35   } // end main
```

```
The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Fig. 9.5 | Program to test class `Time`. (Part 2 of 2.)

9.6 Time Class: Separating Interface from Implementation

- Line 5 of Fig. 9.5 includes the `Time.h` header file to ensure that `Time` objects are created and manipulated correctly.
- Before executing this program, the source-code files in Figs. 9.4 and 9.5 must both be compiled, then linked together—a job performed by the linker.
  - Each source-code is compiled into its object code (`*.obj`) separately.
  - The linker combines all object files into a single executable file.

9.6 Time Class: Separating Interface from Implementation

- The name of the `Time.h` header file in line 5 of Fig. 9.5 is enclosed in quotes (`" "`) rather than angle brackets (`< >`).
- When the preprocessor encounters a header file name in quotes, it attempts to locate the header file in the same directory as the file in which the `#include` directive appears.
- When the preprocessor encounters a header file name in angle brackets (e.g., `<iostream>`), it assumes that the header is part of the C++ Standard Library.
  - Does not look in the directory of the program that's being preprocessed.

9.6 Time Class: Separating Interface from Implementation

- A header file such as `Time.h` (Fig. 9.3) cannot be used to begin program execution, because it does not contain a `main` function.
- If you try to compile and link `Time.h` by itself to create an executable application, Microsoft Visual C++ 2008 produces the linker error message:
  - `error LNK2001: unresolved external symbol _mainCRTStartup`
- To compile and link with GNU C++ on Linux, you must first include the header file in a `.cpp` source-code file, then GNU C++ produces a linker error message containing:
  - `undefined reference to 'main'`
- This error indicates that the linker could not locate the program's `main` function.
- To test class `Time`, you must write a separate source-code file containing a `main` function (such as Fig. 9.5) that instantiates and uses objects of the class.

9.6 Time Class: Separating Interface from Implementation

- The diagram in Fig. 9.6 shows the compilation and linking process that results in an executable `Time` application.
- Often a class's interface and implementation will be created and compiled by one programmer and used by a separate programmer who implements the client code that uses the class.
- So, the diagram shows what's required by both the class-implementation programmer and the client-code programmer.
- The dashed lines in the diagram show the pieces required by the class-implementation programmer, the client-code programmer and the `Time` application user, respectively.

9.6 Time Class: Separating Interface from Implementation

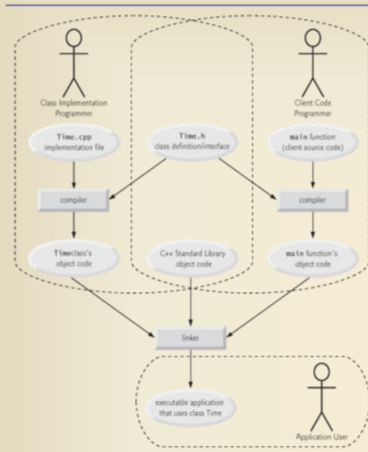

Fig. 9.6 | Compilation and linking process that produces an executable application.

9.6 Time Class: Separating Interface from Implementation

- To create the executable `Time` application, the last step is to link
  - the object code for the `main` function (i.e., the client code),
  - the object code for class `Time`'s member-function implementations
  - the C++ Standard Library object code for the C++ library features (e.g., `cin` and `cout`) used by the class-implementation programmer and the client-code programmer.
- The linker's output is the executable `Time` application.
- Compilers and IDEs typically invoke the linker for you after compiling your code.

9.6 Time Class: Separating Interface from Implementation

Error-Prevention Tip 9.4

To ensure that the preprocessor can locate header files correctly, `#include` preprocessor directives should place the names of user-defined header files in quotes (e.g., `"Time.h"`) and place the names of C++ Standard Library header files in angle brackets (e.g., `<iostream>`).

9.7 Access Functions and Utility Functions

- Access functions can read or display data.
- Another common use for access functions is to test the truth or falsity of conditions—such functions are often called predicate functions.
- An example of a predicate function would be an `isEmpty` function for any container class—a class capable of holding many objects, like a `vector`.
- Useful predicate functions for our `Time` class might be `isAM` and `isPM`.

9.7 Access Functions and Utility Functions

- The program of Figs. 9.7–9.9 demonstrates the notion of a utility function (also called a helper function).
- A utility function is a `private` member function that supports the operation of the class's `public` member functions.
  - It's not part of a class's `public` interface.
- Utility functions are not intended to be used by clients of a class
  - But can be used by friends of a class, as we'll see in Chapter 10.

9.7 Access Functions and Utility Functions

```
1  // Fig. 9.7: SalesPerson.h
2  // SalesPerson class definition.
3  // Member functions defined in SalesPerson.cpp.
4  #ifndef SALESP_H
5  #define SALESP_H
6
7  class SalesPerson
8  {
9  public:
10    static const int monthsPerYear = 12; // months in one year
11    SalesPerson(); // constructor
12    void getSalesFromUser(); // input sales from keyboard
13    void setSales( int, double ); // set sales for a specific month
14    void printAnnualSales(); // summarize and print sales
15 private:
16    double totalAnnualSales(); // prototype for utility function
17    double sales[ monthsPerYear ]; // 12 monthly sales figures
18 }; // end class SalesPerson
19
20 #endif
```

Fig. 9.7 | `SalesPerson` class definition.

9.7 Access Functions and Utility Functions

- Class `SalesPerson` (Fig. 9.7) declares an array of 12 monthly sales figures (line 17) and the class's member functions.
- In Fig. 9.8, the `SalesPerson` constructor (lines 9–13) initializes array `sales` to zero.
- The `public` member function `setSales` (lines 30–37) sets the sales figure for one month in array `sales`.
- The `public` member function `printAnnualSales` (lines 40–45) prints the total sales for the last 12 months.
- The `private` utility function `totalAnnualSales` (lines 48–56) totals the 12 monthly sales figures for the benefit of `printAnnualSales`.
- Member function `printAnnualSales` edits the sales figures into monetary format.

9.7 Access Functions and Utility Functions

```cpp
// Fig. 9.8: SalesPerson.cpp
// SalesPerson class member-function definitions.
#include <iostream>
#include <iomanip>
#include "SalesPerson.h" // include SalesPerson class definition
using namespace std;

// initialize elements of array sales to 0.0
SalesPerson::SalesPerson()
{
   for ( int i = 0; i < monthsPerYear; i++ )
      sales[ i ] = 0.0;
} // end SalesPerson constructor

```

Fig. 9.8 | SalesPerson class member-function definitions. (Part 1 of 3.)

9.7 Access Functions and Utility Functions

```cpp
// get 12 sales figures from the user at the keyboard
void SalesPerson::getSalesFromUser()
{
   double salesFigure;

   for ( int i = 1; i <= monthsPerYear; i++ )
   {
      cout << "Enter sales amount for month " << i << ": ";
      cin >> salesFigure;
      setSales( i, salesFigure );
   } // end for
} // end function getSalesFromUser

// set one of the 12 monthly sales figures; function subtracts
// one from month value for proper subscript in sales array
void SalesPerson::setSales( int month, double amount )
{
   // test for valid month and amount values
   if ( month >= 1 && month <= monthsPerYear && amount > 0 )
      sales[ month - 1 ] = amount; // adjust for subscripts 0-11
   else // invalid month or amount value
      cout << "Invalid month or sales figure" << endl;
} // end function setSales
```

Fig. 9.8 | SalesPerson class member-function definitions. (Part 2 of 3.)

9.7 Access Functions and Utility Functions

```
// print total annual sales (with the help of utility function)
void SalesPerson::printAnnualSales()
{
   cout << setprecision( 2 ) << fixed
      << "\nThe total annual sales are: $"
      << totalAnnualSales() << endl; // call utility function
} // end function printAnnualSales

// private utility function to total annual sales
double SalesPerson::totalAnnualSales()
{
   double total = 0.0; // initialize total

   for ( int i = 0; i < monthsPerYear; i++ ) // summarize sales results
      total += sales[ i ]; // add month i sales to total

   return total;
} // end function totalAnnualSales
```

Fig. 9.8 | SalesPerson class member-function definitions. (Part 3 of 3.)

9.7 Access Functions and Utility Functions

```
// Fig. 9.9: fig09_09.cpp
// Utility function demonstration.
// Compile this program with SalesPerson.cpp

// include SalesPerson class definition from SalesPerson.h
#include "SalesPerson.h"

int main()
{
   SalesPerson s; // create SalesPerson object s

   s.getSalesFromUser(); // note simple sequential code; there are
   s.printAnnualSales(); // no control statements in main
} // end main
```

9.7 Access Functions and Utility Functions

```
Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

The total annual sales are: $60120.59
```


9.7 Access Functions and Utility Functions

- The application's `main` function includes only a simple sequence of member-function calls.
  - There are no control statements.
- The logic of manipulating the `sales` array is completely encapsulated in class `SalesPerson`'s member functions.

Software Engineering Observation 9.9

A phenomenon of object-oriented programming is that once a class is defined, creating and manipulating objects of that class often involve issuing only a simple sequence of member-function calls—few, if any, control statements are needed. By contrast, it's common to have control statements in the implementation of a class's member functions.

9.8 Time Class: Constructors with Default Arguments

- This example enhances class `Time` to demonstrate how arguments are implicitly passed to a constructor.
- Like other functions, constructors can specify default arguments.
- Line 13 of Fig. 9.10 declares the `Time` constructor to include default arguments (zero for each argument).
- In Fig. 9.11, line 12 of the constructor calls member function `setTime` with the values passed to the constructor (or the default values).
- If a value is out of range, that value is set to zero (to ensure that each data member remains in a consistent state).

9.8 Time Class: Constructors with Default Arguments

```
1  // Fig. 9.10: Time.h
2  // Time class containing a constructor with default arguments.
3  // Member functions defined in Time.cpp.
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME_H
7  #define TIME_H
8
9  // Time abstract data type definition
10 class Time
11 {
12 public:
13    Time( int = 0, int = 0, int = 0 ); // default constructor
14    void setTime( int, int, int ); // set hour, minute, second
15    void printUniversal(); // output time in universal-time format
16    void printStandard(); // output time in standard-time format
17 private:
18    int hour; // 0 - 23 (24-hour clock format)
19    int minute; // 0 - 59
20    int second; // 0 - 59
21 }; // end class Time
22
23 #endif
```

Fig. 9.10 | `Time` class containing a constructor with default arguments.

9.8 Time Class: Constructors with Default Arguments

```cpp
// Fig.9.11: Time.cpp
// Member-function definitions for class Time.
#include <iostream>
#include <iomanip>
#include "Time.h" // include definition of class Time from Time.h
using namespace std;

// Time constructor initializes each data member to zero;
// ensures that Time objects start in a consistent state
Time::Time( int hr, int min, int sec )
{
   setTime( hr, min, sec ); // validate and set time
} // end Time constructor

// set new Time value using universal time; ensure that
// the data remains consistent by setting invalid values to zero
void Time::setTime( int h, int m, int s )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
   minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
   second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
} // end function setTime
```

Fig. 9.11 | Time class member-function definitions including a constructor that takes arguments. (Part 1 of 2.)

9.8 Time Class: Constructors with Default Arguments

```cpp

// print Time in universal-time format (HH:MM:SS)
void Time::printUniversal()
{
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
      << setw( 2 ) << minute << ":" << setw( 2 ) << second;
} // end function printUniversal

// print Time in standard-time format (HH:MM:SS AM or PM)
void Time::printStandard()
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":"
      << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 )
      << second << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
```

Fig. 9.11 | Time class member-function definitions including a constructor that takes arguments. (Part 2 of 2.)

9.8 Time Class: Constructors with Default Arguments

- Function `main` in Fig. 9.12 initializes five `Time` objects—one with all three arguments defaulted in the implicit constructor call (line 9), one with one argument specified (line 10), one with two arguments specified (line 11), one with three arguments specified (line 12) and one with three invalid arguments specified (line 13).
- Then the program displays each object in universal-time and standard-time formats.

9.8 Time Class: Constructors with Default Arguments

```cpp
1  // Fig. 9.12: fig09_12.cpp
2  // Demonstrating a default constructor for class Time.
3  #include <iostream>
4  #include "Time.h" // include definition of class Time from Time.h
5  using namespace std;
6
7  int main()
8  {
9     Time t1; // all arguments defaulted
10    Time t2( 2 ); // hour specified; minute and second defaulted
11    Time t3( 21, 34 ); // hour and minute specified; second defaulted
12    Time t4( 12, 25, 42 ); // hour, minute and second specified
13    Time t5( 27, 74, 99 ); // all bad values specified
14
15    cout << "Constructed with:\n\nt1: all arguments defaulted\n  ";
16    t1.printUniversal(); // 00:00:00
17    cout << "\n  ";
18    t1.printStandard(); // 12:00:00 AM
19
20    cout << "\n\nt2: hour specified; minute and second defaulted\n  ";
21    t2.printUniversal(); // 02:00:00
22    cout << "\n  ";
23    t2.printStandard(); // 2:00:00 AM
```

Fig. 9.12 | Constructor with default arguments. (Part 1 of 3.)

9.8 Time Class: Constructors with Default Arguments

```
    cout << "\n\nt3: hour and minute specified; second defaulted\n  ";
    t3.printUniversal(); // 21:34:00
    cout << "\n  ";
    t3.printStandard(); // 9:34:00 PM

    cout << "\n\nt4: hour, minute and second specified\n  ";
    t4.printUniversal(); // 12:25:42
    cout << "\n  ";
    t4.printStandard(); // 12:25:42 PM

    cout << "\n\nt5: all invalid values specified\n  ";
    t5.printUniversal(); // 00:00:00
    cout << "\n  ";
    t5.printStandard(); // 12:00:00 AM
    cout << endl;
} // end main
```

Fig. 9.12 | Constructor with default arguments. (Part 2 of 3.)

9.8 Time Class: Constructors with Default Arguments

```
Constructed with:

t1: all arguments defaulted
  00:00:00
  12:00:00 AM

t2: hour specified; minute and second defaulted
  02:00:00
  2:00:00 AM

t3: hour and minute specified; second defaulted
  21:34:00
  9:34:00 PM

t4: hour, minute and second specified
  12:25:42
  12:25:42 PM

t5: all invalid values specified
  00:00:00
  12:00:00 AM
```

Fig. 9.12 | Constructor with default arguments. (Part 3 of 3.)

9.8 Time Class: Constructors with Default Arguments

- Any constructor that takes no arguments is called a default constructor.
- A class gets a default constructor in one of two ways:
- The compiler implicitly creates a default constructor in a class that does not define a constructor.
  - Such a constructor does not initialize the class's data members, but does call the default constructor for each data member that's an object of another class.
- You explicitly define a constructor that takes no arguments.
  - Such a default constructor will call the default constructor for each data member that's an object of another class and will perform additional initialization specified by you.
- If you define a constructor with arguments, C++ will not implicitly create a default constructor for that class.

9.8 Time Class: Constructors with Default Arguments

Software Engineering Observation 9.10

Any change to the default argument values of a function requires the client code to be recompiled (to ensure that the program still functions correctly).

Software Engineering Observation 9.11

Data members can be initialized in a constructor, or their values may be set later after the object is created. However, it's a good software engineering practice to ensure that an object is fully initialized before the client code invokes the object's member functions. You should not rely on the client code to ensure that an object gets initialized properly.

9.8 Time Class: Constructors with Default Arguments

Error-Prevention Tip 9.5

Unless no initialization of your class's data members is necessary (almost never), provide a constructor to ensure that your class's data members are initialized with meaningful values when each new object of your class is created.

9.9 Destructors

- The name of the destructor for a class is the tilde character (~) followed by the class name.
  - Another type of special member function.
- The destructor is called implicitly when an object is destroyed.
  - For example, an automatic object is destroyed when program execution leaves its scope in which that object was instantiated.
- A destructor receives no parameters and returns no value.
  - May not specify a return type—not even `void`.
- A class may have only one destructor.
- A destructor must be `public`.
- If you do not explicitly provide a destructor, the compiler creates an "empty" destructor.

9.10 When Constructors and Destructors Are Called

- Constructors and destructors are called implicitly by the compiler.
- The order in which these function calls occur depends on the order in which execution enters and leaves the scopes where the objects are instantiated.
- Generally, destructor calls are made in the reverse order of the corresponding constructor calls
- The storage classes of objects can alter the order in which destructors are called.

9.10 When Constructors and Destructors Are Called

- Constructors are called for objects defined in global scope before any other function (including `main`) begins execution.
- The corresponding destructors are called when `main` terminates.
- The constructor for an automatic local object is called when execution reaches the point where that object is defined
- The corresponding destructor is called when the block in which that object is defined has finished executing.
- The constructor for a `static` local object is called only once, when execution first reaches the point where the object is defined
- The corresponding destructor is called when `main` terminates or the program calls function `exit`.

9.10 When Constructors and Destructors Are Called

```cpp
// Fig. 9.13: CreateAndDestroy.h
// CreateAndDestroy class definition.
// Member functions defined in CreateAndDestroy.cpp.
#include <string>
using namespace std;

#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy
{
public:
   CreateAndDestroy( int, string ); // constructor
   ~CreateAndDestroy(); // destructor
private:
   int objectID; // ID number for object
   string message; // message describing object
}; // end class CreateAndDestroy

#endif
```

Fig. 9.13 | CreateAndDestroy class definition.

9.10 When Constructors and Destructors Are Called

```cpp
// Fig. 9.14: CreateAndDestroy.cpp
// CreateAndDestroy class member-function definitions.
#include <iostream>
#include "CreateAndDestroy.h"// include CreateAndDestroy class definition
using namespace std;

// constructor
CreateAndDestroy::CreateAndDestroy( int ID, string messageString )
{
   objectID = ID; // set object's ID number
   message = messageString; // set object's descriptive message

   cout << "Object " << objectID << "   constructor runs   "
      << message << endl;
} // end CreateAndDestroy constructor

// destructor
CreateAndDestroy::~CreateAndDestroy()
{
   // output newline for certain objects; helps readability
   cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );

   cout << "Object " << objectID << "   destructor runs    "
      << message << endl;
} // end ~CreateAndDestroy destructor
```

Fig. 9.14 | CreateAndDestroy class member-function definitions.

9.10 When Constructors and Destructors Are Called

```cpp
// Fig. 9.15: fig09_15.cpp
// Demonstrating the order in which constructors and
// destructors are called.
#include <iostream>
#include "CreateAndDestroy.h" // include CreateAndDestroy class definition
using namespace std;

void create( void ); // prototype

CreateAndDestroy first( 1, "(global before main)" ); // global object

int main()
{
   cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
   CreateAndDestroy second( 2, "(local automatic in main)" );
   static CreateAndDestroy third( 3, "(local static in main)" );

   create(); // call function to create objects

   cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
   CreateAndDestroy fourth( 4, "(local automatic in main)" );
   cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
} // end main
```

Fig. 9.15 | Order in which constructors and destructors are called. (Part 1 of 3.)

9.10 When Constructors and Destructors Are Called

```cpp

// function to create objects
void create( void )
{
   cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
   CreateAndDestroy fifth( 5, "(local automatic in create)" );
   static CreateAndDestroy sixth( 6, "(local static in create)" );
   CreateAndDestroy seventh( 7, "(local automatic in create)" );
   cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
} // end function create
```

Fig. 9.15 | Order in which constructors and destructors are called. (Part 2 of 3.)

9.10 When Constructors and Destructors Are Called

```
Object 1   constructor runs   (global before main)

MAIN FUNCTION: EXECUTION BEGINS
Object 2   constructor runs   (local automatic in main)
Object 3   constructor runs   (local static in main)

CREATE FUNCTION: EXECUTION BEGINS
Object 5   constructor runs   (local automatic in create)
Object 6   constructor runs   (local static in create)
Object 7   constructor runs   (local automatic in create)

CREATE FUNCTION: EXECUTION ENDS
Object 7   destructor runs    (local automatic in create)
Object 5   destructor runs    (local automatic in create)

MAIN FUNCTION: EXECUTION RESUMES
Object 4   constructor runs   (local automatic in main)

MAIN FUNCTION: EXECUTION ENDS
Object 4   destructor runs    (local automatic in main)
Object 2   destructor runs    (local automatic in main)

Object 6   destructor runs    (local static in create)
Object 3   destructor runs    (local static in main)

Object 1   destructor runs    (global before main)
```

Fig. 9.15 | Order in which constructors and destructors are called. (Part 3 of 3.)

9.11 Time Class: Using Set and Get Functions

- Classes often provide `public` member functions to allow clients of the class to ***set*** (i.e., assign values to) or ***get*** (i.e., obtain the values of) `private` data members.
  - These member function names need not begin with `set` or `get`, but this naming convention is common.
- This example enhances class `Time` to include public functions that allow the client code to *set* and *get* the values of the `private` data members `hour`, `minute` and `second`.
- The *set* functions (defined at lines 25–28, 31–34 and 37–40) strictly control the setting of the data members.
- Each *get* function (defined at lines 43–46, 49–52 and 55–58) simply returns the appropriate data member's value.

9.11 Time Class: Using Set and Get Functions

```
// Fig. 9.16: Time.h
// Time class containing a constructor with default arguments.
// Member functions defined in Time.cpp.

// prevent multiple inclusions of header file
#ifndef TIME_H
#define TIME_H

// Time abstract data type definition
class Time
{
public:
   Time( int = 0, int = 0, int = 0 ); // default constructor

   // set functions
   void setTime( int, int, int ); // set hour, minute, second
   void setHour( int ); // set hour (after validation)
   void setMinute( int ); // set minute (after validation)
   void setSecond( int ); // set second (after validation)

```

Fig. 9.16 | Time class containing a constructor with default arguments. (Part 1 of 2.)

9.11 Time Class: Using Set and Get Functions

```
   // get functions
   int getHour(); // return hour
   int getMinute(); // return minute
   int getSecond(); // return second

   void printUniversal(); // output time in universal-time format
   void printStandard(); // output time in standard-time format
private:
   int hour; // 0 - 23 (24-hour clock format)
   int minute; // 0 - 59
   int second; // 0 - 59
}; // end class Time

#endif
```

Fig. 9.16 | Time class containing a constructor with default arguments. (Part 2 of 2.)

9.11 Time Class: Using Set and Get Functions

```cpp
// Fig. 9.17: Time.cpp
// Member-function definitions for class Time.
#include <iostream>
#include <iomanip>
#include "Time.h" // include definition of class Time from Time.h
using namespace std;

// Time constructor initializes each data member to zero;
// ensures that Time objects start in a consistent state
Time::Time( int hr, int min, int sec )
{
   setTime( hr, min, sec ); // validate and set time
} // end Time constructor

// set new Time value using universal time; ensure that
// the data remains consistent by setting invalid values to zero
void Time::setTime( int h, int m, int s )
{
   setHour( h ); // set private field hour
   setMinute( m ); // set private field minute
   setSecond( s ); // set private field second
} // end function setTime
```

Fig. 9.17 | Time class member-function definitions including a constructor that takes arguments. (Part 1 of 4.)

9.11 Time Class: Using Set and Get Functions

```cpp

// set hour value
void Time::setHour( int h )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
} // end function setHour

// set minute value
void Time::setMinute( int m )
{
   minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
} // end function setMinute

// set second value
void Time::setSecond( int s )
{
   second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
} // end function setSecond

```

Fig. 9.17 | Time class member-function definitions including a constructor that takes arguments. (Part 2 of 4.)

9.11 Time Class: Using Set and Get Functions

```cpp
// return hour value
int Time::getHour()
{
   return hour;
} // end function getHour

// return minute value
int Time::getMinute()
{
   return minute;
} // end function getMinute

// return second value
int Time::getSecond()
{
   return second;
} // end function getSecond

```

Fig. 9.17 | `Time` class member-function definitions including a constructor that takes arguments. (Part 3 of 4.)

9.11 Time Class: Using Set and Get Functions

```cpp
// print Time in universal-time format (HH:MM:SS)
void Time::printUniversal()
{
   cout << setfill( '0' ) << setw( 2 ) << getHour() << ":"
      << setw( 2 ) << getMinute() << ":" << setw( 2 ) << getSecond();
} // end function printUniversal

// print Time in standard-time format (HH:MM:SS AM or PM)
void Time::printStandard()
{
   cout << ( ( getHour() == 0 || getHour() == 12 ) ? 12 : getHour() % 12 )
      << ":" << setfill( '0' ) << setw( 2 ) << getMinute()
      << ":" << setw( 2 ) << getSecond() << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
```

Fig. 9.17 | `Time` class member-function definitions including a constructor that takes arguments. (Part 4 of 4.)

9.11 Time Class: Using Set and Get Functions

```
// Fig. 9.18: fig09_18.cpp
// Demonstrating the Time class set and get functions
#include <iostream>
#include "time.h"
using namespace std;

void incrementMinutes( Time &, const int ); // prototype

int main()
{
   Time t; // create Time object

   // set time using individual set functions
   t.setHour( 17 );  // set hour to valid value
   t.setMinute( 34 ); // set minute to valid value
   t.setSecond( 25 ); // set second to valid value

   // use get functions to obtain hour, minute and second
   cout << "Result of setting all valid values:\n"
        << "  Hour: " << t.getHour()
        << "  Minute: " << t.getMinute()
        << "  Second: " << t.getSecond();
```

Fig. 9.18 | *Set* and *get* functions manipulating an object's `private` data. (Part 1 of 4.)

9.11 Time Class: Using Set and Get Functions

```

   // set time using individual set functions
   t.setHour( 234 ); // invalid hour set to 0
   t.setMinute( 43 ); // set minute to valid value
   t.setSecond( 6373 ); // invalid second set to 0

   // display hour, minute and second after setting
   // invalid hour and second values
   cout << "\n\nResult of attempting to set invalid hour and"
        << " second:\n  Hour: " << t.getHour()
        << "  Minute: " << t.getMinute()
        << "  Second: " << t.getSecond() << "\n\n";

   t.setTime( 11, 58, 0 ); // set time
   incrementMinutes( t, 3 ); // increment t's minute by 3
} // end main

```

Fig. 9.18 | *Set* and *get* functions manipulating an object's `private` data. (Part 2 of 4.)

9.11 Time Class: Using Set and Get Functions

```cpp
// add specified number of minutes to a Time object
void incrementMinutes( Time &tt, const int count )
{
   cout << "Incrementing minute " << count
        << " times:\nStart time: ";
   tt.printStandard();

   for ( int i = 0; i < count; i++ ) {
      tt.setMinute( ( tt.getMinute() + 1 ) % 60 );

      if ( tt.getMinute() == 0 )
         tt.setHour( ( tt.getHour() + 1 ) % 24);

      cout << "\nminute + 1: ";
      tt.printStandard();
   } // end for

   cout << endl;
} // end function incrementMinutes
```

Fig. 9.18 | *Set* and *get* functions manipulating an object's `private` data. (Part 3 of 4.)

9.11 Time Class: Using Set and Get Functions

```
Result of setting all valid values:
  Hour: 17  Minute: 34  Second: 25

Result of attempting to set invalid hour and second:
  Hour: 0  Minute: 43  Second: 0

Incrementing minute 3 times:
Start time: 11:58:00 AM
minute + 1: 11:59:00 AM
minute + 1: 12:00:00 PM
minute + 1: 12:01:00 PM
```

Fig. 9.18 | *Set* and *get* functions manipulating an object's `private` data. (Part 4 of 4.)

9.11 Time Class: Using Set and Get Functions

- Declaring data members with access specifier `private` enforces data hiding.
- Providing `public` *set* and *get* functions allows clients of a class to access the hidden data, but only indirectly.
- The client knows that it's attempting to modify or obtain an object's data, but the client does not know how the object performs these operations.
- In some cases, a class may internally represent a piece of data one way, but expose that data to clients in a different way.
- The *set* and *get* functions allow a client to interact with an object, but the object's `private` data remains safely encapsulated (i.e., hidden) in the object itself.

9.11 Time Class: Using Set and Get Functions

- `Time`'s *set* and *get* functions are called throughout the class's body.
- Consider changing the representation of the time from three `int` values to a single `int` value
  - Represent the total number of seconds elapsed since midnight.
- Only the bodies of the functions that access the `private` data directly would need to change.
  - Those *set* and *get* functions for the `hour`, `minute` and `second`.
- There would be no need to modify the bodies of functions `setTime`, `printUniversal` or `printStandard`.

9.11 Time Class: Using Set and Get Functions

- Designing the class in this manner reduces the likelihood of programming errors when altering the class's implementation.
- Similarly, the `Time` constructor could be written to include a copy of the appropriate statements from function `setTime`.
- Doing so may be slightly more efficient, because the extra constructor call and call to `setTime` are eliminated.
- However, duplicating statements in multiple functions or constructors makes changing the class's internal data representation more difficult.
- Having the `Time` constructor call function `setTime` directly requires any changes to the implementation of `setTime` to be made only once.

9.11 Time Class: Using Set and Get Functions

Software Engineering Observation 9.12

If a member function of a class already provides all or part of the functionality required by a constructor (or other member function) of the class, call that member function from the constructor (or other member function). This simplifies the maintenance of the code and reduces the likelihood of an error if the implementation of the code is modified. As a general rule: Avoid repeating code.

Good Programming Practice 9.3

Always try to localize the effects of changes to a class's data members by accessing and manipulating the data members through their get *and* set *functions. Changes to the name of a data member or the data type used to store a data member then affect only the corresponding* get *and* set *functions, but not the callers of those functions.*

9.12 Time Class: A Subtle Trap—Returning a Reference to a private Data Member

- A reference to an object is an alias for the name of the object and, hence, may be used on the left side of an assignment statement.
  - In this context, the reference makes a perfectly acceptable *lvalue* that can receive a value.
- One way to use this capability (unfortunately!) is to have a `public` member function of a class return a reference to a `private` data member of that class.
- If a function returns a `const` reference, that reference cannot be used as a modifiable *lvalue*.

9.12 Time Class: A Subtle Trap—Returning a Reference to a private Data Member

- The program of Figs. 9.19–9.21 uses a simplified `Time` class (Fig. 9.19 and Fig. 9.20) to demonstrate returning a reference to a `private` data member with member function `badSetHour` (declared in Fig. 9.19 in line 15 and defined in Fig. 9.20 in lines 27–31).
- Such a reference return actually makes a call to member function `badSetHour` an alias for `private` data member `hour`!
- The function call can be used in any way that the `private` data member can be used, including as an *lvalue* in an assignment statement, thus enabling clients of the class to clobber the class's private data at will!
- The same problem would occur if a pointer to the `private` data were to be returned by the function.

9.12 Time Class: A Subtle Trap—Returning a Reference to a private Data Member

```
// Fig. 9.19: Time.h
// Time class declaration.
// Member functions defined in Time.cpp

// prevent multiple inclusions of header file
#ifndef TIME_H
#define TIME_H

class Time
{
public:
   Time( int = 0, int = 0, int = 0 );
   void setTime( int, int, int );
   int getHour();
   int &badSetHour( int ); // DANGEROUS reference return
private:
   int hour;
   int minute;
   int second;
}; // end class Time

#endif
```

Fig. 9.19 | Time class declaration.

9.12 Time Class: A Subtle Trap—Returning a Reference to a private Data Member

```
// Fig. 9.20: Time.cpp
// Time class member-function definitions.
#include "Time.h" // include definition of class Time

// constructor function to initialize private data; calls member function
// setTime to set variables; default values are 0 (see class definition)
Time::Time( int hr, int min, int sec )
{
   setTime( hr, min, sec );
} // end Time constructor

// set values of hour, minute and second
void Time::setTime( int h, int m, int s )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
   minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
   second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
} // end function setTime

// return hour value
int Time::getHour()
{
   return hour;
} // end function getHour
```

Fig. 9.20 | Time class member-function definitions. (Part 1 of 2.)

9.12 Time Class: A Subtle Trap—Returning a Reference to a private Data Member

```cpp

// POOR PRACTICE: Returning a reference to a private data member.
int &Time::badSetHour( int hh )
{
   hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
   return hour; // DANGEROUS reference return
} // end function badSetHour
```

Fig. 9.20 | Time class member-function definitions. (Part 2 of 2.)

9.12 Time Class: A Subtle Trap—Returning a Reference to a private Data Member

```cpp
// Fig. 9.21: fig09_21.cpp
// Demonstrating a public member function that
// returns a reference to a private data member.
#include <iostream>
#include "Time.h" // include definition of class Time
using namespace std;

int main()
{
   Time t; // create Time object

   // initialize hourRef with the reference returned by badSetHour
   int &hourRef = t.badSetHour( 20 ); // 20 is a valid hour

   cout << "Valid hour before modification: " << hourRef;
   hourRef = 30; // use hourRef to set invalid value in Time object t
   cout << "\nInvalid hour after modification: " << t.getHour();

   // Dangerous: Function call that returns
   // a reference can be used as an lvalue!
   t.badSetHour( 12 ) = 74; // assign another invalid value to hour

```

Fig. 9.21 | Returning a reference to a private data member. (Part 1 of 2.)

9.12 Time Class: A Subtle Trap—Returning a Reference to a private Data Member

```
    cout << "\n\n*************************************************\n"
       << "POOR PROGRAMMING PRACTICE!!!!!!!!\n"
       << "t.badSetHour( 12 ) as an lvalue, invalid hour: "
       << t.getHour()
       << "\n*************************************************" << endl;
} // end main
```

```
Valid hour before modification: 20
Invalid hour after modification: 30

*************************************************
POOR PROGRAMMING PRACTICE!!!!!!!!
t.badSetHour( 12 ) as an lvalue, invalid hour: 74
*************************************************
```

Fig. 9.21 | Returning a reference to a `private` data member. (Part 2 of 2.)

9.13 Default Memberwise Assignment

- The assignment operator (=) can be used to assign an object to another object of the same type.
- By default, such assignment is performed by memberwise assignment
  - Each data member of the RHS object is assigned individually to the same data member in the LHS object.
- Caution: *Memberwise assignment can cause serious problems when used with a class whose data members contain pointers to dynamically allocated memory; we discuss these problems in Chapter 11 and show how to deal with them.*

9.13 Default Memberwise Assignment

```cpp
// Fig. 9.19: Date.h
// Date class declaration.  Member functions are defined in Date.cpp.

// prevent multiple inclusions of header file
#ifndef DATE_H
#define DATE_H

// class Date definition
class Date
{
public:
   Date( int = 1, int = 1, int = 2000 ); // default constructor
   void print();
private:
   int month;
   int day;
   int year;
}; // end class Date

#endif
```

Fig. 9.22 | Date class declaration.

9.13 Default Memberwise Assignment

```cpp
// Fig. 9.20: Date.cpp
// Date class member-function definitions.
#include <iostream>
#include "Date.h" // include definition of class Date from Date.h
using namespace std;

// Date constructor (should do range checking)
Date::Date( int m, int d, int y )
{
   month = m;
   day = d;
   year = y;
} // end constructor Date

// print Date in the format mm/dd/yyyy
void Date::print()
{
   cout << month << '/' << day << '/' << year;
} // end function print
```

Fig. 9.23 | Date class member-function definitions.

9.13 Default Memberwise Assignment

```cpp
// Fig. 9.21: fig09_21.cpp
// Demonstrating that class objects can be assigned
// to each other using default memberwise assignment.
#include <iostream>
#include "Date.h" // include definition of class Date from Date.h
using namespace std;

int main()
{
   Date date1( 7, 4, 2004 );
   Date date2; // date2 defaults to 1/1/2000

   cout << "date1 = ";
   date1.print();
   cout << "\ndate2 = ";
   date2.print();

   date2 = date1; // default memberwise assignment

   cout << "\n\nAfter default memberwise assignment, date2 = ";
   date2.print();
   cout << endl;
} // end main
```

Fig. 9.24 | Default memberwise assignment. (Part 1 of 2.)

9.13 Default Memberwise Assignment

```
date1 = 7/4/2004
date2 = 1/1/2000

After default memberwise assignment, date2 = 7/4/2004
```

Fig. 9.24 | Default memberwise assignment. (Part 2 of 2.)

9.13 Default Memberwise Assignment

- Objects may be passed as function arguments and may be returned from functions.
- Such passing and returning is performed using pass-by-value by default—a copy of the object is passed or returned.
  - Time-consuming for large objects.
- In such cases, C++ creates a new object and uses a copy constructor to copy the original object's values into the new object.
- For each class, the compiler provides a default copy constructor that copies each member of the original object into the corresponding member of the new object.

9.13 Default Memberwise Assignment

Performance Tip 9.3

Passing an object by value is good from a security standpoint, because the called function has no access to the original object in the caller, but pass-by-value can degrade performance when making a copy of a large object. An object can be passed by reference by passing either a pointer or a reference to the object. Pass-by-reference offers good performance but is weaker from a security standpoint, because the called function is given access to the original object. Pass-by-`const`*-reference is a safe, good-performing alternative (this can be implemented with a* `const` *reference parameter or with a pointer-to-*`const`*-data parameter).*

Summary

- Concepts of object-oriented programming
- Classes, Objects, Member Functions and Data Members
- Public and Private
- Constructor / Destructor
- Separating Interface from Implementation
- Access Functions and Utility Functions
- Using Set and Get Functions
- Trap—Returning a Reference to a private Data Member
- Default Memberwise Assignment

Before We End This Chapter…

22.6 Example: Card Shuffling and Dealing Simulation

- The card shuffling and dealing program in Figs. 22.2–22.4 is similar to the one described in Exercise 10.10.
- This program represents the deck of cards as a `vector` of structures and uses high-performance shuffling and dealing algorithms.
- The constructor (lines 12–32 of Fig. 22.3) initializes the `Card` `vector` in order with character strings representing Ace through King of each suit.

22.6 Example: Card Shuffling and Dealing Simulation

- Function `shuffle` implements the high-performance shuffling algorithm.
  - The function loops through all 52 cards (subscripts 0 to 51).
  - For each card, a number between 0 and 51 is picked randomly.
  - Next, the current `Card` structure and the randomly selected `Card` structure are swapped in the `vector`.
  - A total of 52 swaps are made in a single pass of the entire `vector`, and the `vector` of `Card` structures is shuffled.
  - Because the `Card` structures were swapped in place in the `vector`, the dealing algorithm implemented in function `deal` requires only one pass of the `vector` to deal the shuffled cards.

22.6 Example: Card Shuffling and Dealing Simulation

```cpp
// Fig. 22.2: DeckOfCards.h
// Definition of class DeckOfCards that
// represents a deck of playing cards.
#include <string>
#include <vector>
using namespace std;

// Card structure definition
struct Card
{
   string face;
   string suit;
}; // end structure Card

```

Fig. 22.2 | Header file for `DeckOfCards` class. (Part 1 of 2.)

22.6 Example: Card Shuffling and Dealing Simulation

```cpp
// DeckOfCards class definition
class DeckOfCards
{
public:
   static const int numberOfCards = 52;
   static const int faces = 13;
   static const int suits = 4;

   DeckOfCards(); // constructor initializes deck
   void shuffle(); // shuffles cards in deck
   void deal() const; // deals cards in deck

private:
   vector< Card > deck; // represents deck of cards
}; // end class DeckOfCards
```

Fig. 22.2 | Header file for `DeckOfCards` class. (Part 2 of 2.)

22.6 Example: Card Shuffling and Dealing Simulation

```cpp
// Fig. 22.3: DeckOfCards.cpp
// Member-function definitions for class DeckOfCards that simulates
// the shuffling and dealing of a deck of playing cards.
#include <iostream>
#include <iomanip>
#include <cstdlib> // prototypes for rand and srand
#include <ctime> // prototype for time
#include "DeckOfCards.h" // DeckOfCards class definition
using namespace std;

// no-argument DeckOfCards constructor intializes deck
DeckOfCards::DeckOfCards()
   : deck( numberOfCards )
{
   // initialize suit array
   static string suit[ suits ] =
      { "Hearts", "Diamonds", "Clubs", "Spades" };

   // initialize face array
   static string face[ faces ] =
      { "Ace", "Deuce", "Three", "Four", "Five", "Six", "Seven",
      "Eight", "Nine", "Ten", "Jack", "Queen", "King" };

```

Fig. 22.3 | Class file for `DeckOfCards`. (Part 1 of 3.)

22.6 Example: Card Shuffling and Dealing Simulation

```cpp
   // set values for deck of 52 Cards
   for ( int i = 0; i < numberOfCards; i++ )
   {
      deck[ i ].face = face[ i % faces ];
      deck[ i ].suit = suit[ i / faces ];
   } // end for

   srand( time( 0 ) ); // seed random number generator
} // end no-argument DeckOfCards constructor

// shuffle cards in deck
void DeckOfCards::shuffle()
{
   // shuffle cards randomly
   for ( int i = 0; i < numberOfCards; i++ )
   {
      int j = rand() % numberOfCards;
      Card temp = deck[ i ];
      deck[ i ] = deck[ j ];
      deck[ j ] = temp;
   } // end for
} // end function shuffle

```

Fig. 22.3 | Class file for `DeckOfCards`. (Part 2 of 3.)

22.6 Example: Card Shuffling and Dealing Simulation

```cpp
// deal cards in deck
void DeckOfCards::deal() const
{
   // display each card's face and suit
   for ( int i = 0; i < numberOfCards; i++ )
      cout << right << setw( 5 ) << deck[ i ].face << " of "
         << left << setw( 8 ) << deck[ i ].suit
         << ( ( i + 1 ) % 2 ? '\t' : '\n' );
} // end function deal
```

Fig. 22.3 | Class file for `DeckOfCards`. (Part 3 of 3.)

22.6 Example: Card Shuffling and Dealing Simulation

```cpp
// Fig. 22.4: fig22_04.cpp
// Card shuffling and dealing program.
#include "DeckOfCards.h" // DeckOfCards class definition

int main()
{
   DeckOfCards deckOfCards; // create DeckOfCards object
   deckOfCards.shuffle(); // shuffle the cards in the deck
   deckOfCards.deal(); // deal the cards in the deck
} // end main
```

Fig. 22.4 | High-performance card shuffling and dealing simulation. (Part 1 of 2.)

22.6 Example: Card Shuffling and Dealing Simulation

```
 King of Clubs          Ten of Diamonds
 Five of Diamonds      Jack of Clubs
Seven of Spades        Five of Clubs
Three of Spades        King of Hearts
  Ten of Clubs        Eight of Spades
Eight of Hearts          Six of Hearts
 Nine of Diamonds      Nine of Clubs
Three of Diamonds     Queen of Hearts
  Six of Clubs        Seven of Hearts
Seven of Diamonds      Jack of Diamonds
 Jack of Spades        King of Diamonds
Deuce of Diamonds      Four of Clubs
Three of Clubs         Five of Hearts
Eight of Clubs          Ace of Hearts
Deuce of Spades         Ace of Clubs
  Ten of Spades       Eight of Diamonds
  Ten of Hearts          Six of Spades
Queen of Diamonds      Nine of Hearts
Seven of Clubs        Queen of Clubs
Deuce of Clubs        Queen of Spades
Three of Hearts        Five of Spades
Deuce of Hearts        Jack of Hearts
 Four of Hearts         Ace of Diamonds
 Nine of Spades        Four of Diamonds
  Ace of Spades         Six of Diamonds
 Four of Spades        King of Spades
```

Fig. 22.4 | High-performance card shuffling and dealing simulation. (Part 2 of 2.)

Andy, Yu-Guang Chen
Assistant Professor, Department of EE, NCU
Email: andyygchen@ee.ncu.edu.tw