CHAPTER

9

OBJECTS AND CLASSES

Objectives

- To describe objects and classes, and to use classes to model objects (§9.2).
- To use UML graphical notations to describe classes and objects (§9.2).
- To demonstrate defining classes and creating objects (§9.3).
- To create objects using constructors (§9.4).
- To access data fields and invoke functions using the object member access operator (`.`) (§9.5).
- To separate a class definition from a class implementation (§9.6).
- To prevent multiple inclusions of header files using the `#ifndef` inclusion guard directive (§9.7).
- To know what inline functions in a class are (§9.8).
- To declare private data fields with appropriate `get` and `set` functions for data field encapsulation and make classes easy to maintain (§9.9).
- To understand the scope of data fields (§9.10).
- To apply class abstraction to develop software (§9.11).

9.1 Introduction

Key Point

Object-oriented programming enables you to develop large-scale software effectively.

why OOP?

Having learned the material in earlier chapters, you are able to solve many programming problems using selections, loops, functions, and arrays. However, these features are not sufficient for developing large-scale software systems. This chapter begins the introduction of object-oriented programming, which will enable you to develop large-scale software systems effectively.

9.2 Defining Classes for Objects

Key Point

A class defines the properties and behaviors for objects.

object-oriented programming
object

Object-oriented programming (OOP) involves programming using objects. An *object* represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique identity, state, and behavior.

state
property
data field

- The *state* of an object (also known as *properties* or *attributes*) is represented by *data fields* with their current values. A circle object, for example, has a data field, `radius`, which is the property that characterizes a circle. A rectangle object, for example, has data fields, `width` and `height`, which are the properties that characterize a rectangle.

behavior

- The *behavior* of an object (also known as *actions*) is defined by functions. To invoke a function on an object is to ask the object to perform an action. For example, you may define a function named `getArea()` for circle objects. A circle object may invoke `getArea()` to return its area.

class
contract
instantiation
object
instance

Objects of the same type are defined using a common class. A *class* is a template, blueprint, or *contract* that defines what an object's data fields and functions will be. An object is an instance of a class. You can create many instances of a class. Creating an instance is referred to as *instantiation*. The terms *object* and *instance* are often interchangeable. The relationship between classes and objects is analogous to the relationship between apple pie recipes and apple pies. You can make as many apple pies as you want from a single recipe. Figure 9.1 shows a class named `Circle` and its three objects.

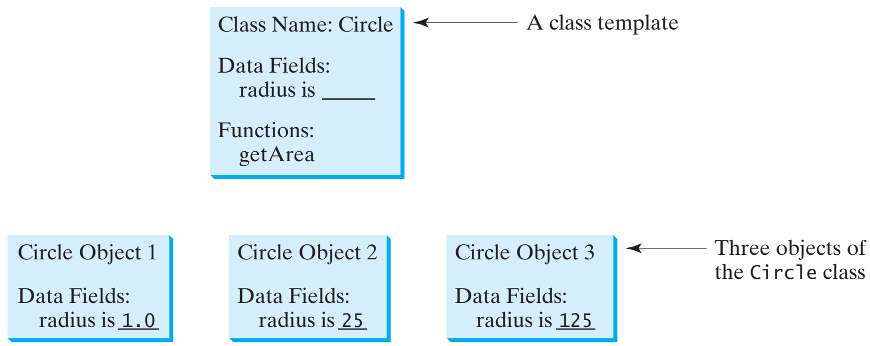

FIGURE 9.1 A class is a blueprint for creating objects.

class
data field
function

A C++ *class* uses variables to define *data fields* and *functions* to define behaviors. Additionally, a class provides functions of a special type, known as *constructors*, which are invoked when a new object is created. A constructor is a special kind of function. Constructors can perform any action, but they are designed to perform initializing actions, such as initializing the data fields of objects. Figure 9.2 shows an example of the class for `Circle` objects.

constructor

```
class Circle
{
public:
  // The radius of this circle
  double radius;

  // Construct a circle object
  Circle()
  {
    radius = 1;
  }

  // Construct a circle object
  Circle(double newRadius)
  {
    radius = newRadius;
  }

  // Return the area of this circle
  double getArea()
  {
    return radius * radius * 3.14159;
  }
};
```

Data field

Constructors

Function

FIGURE 9.2 A class is a blueprint that defines objects of the same type.

The illustration of class and objects in Figure 9.1 can be standardized using UML (Unified Modeling Language) notation, as shown in Figure 9.3. This is called a *UML class diagram*, or simply *class diagram*. The data field is denoted as

UML class diagram

```
dataFieldName: dataFieldType
```

The constructor is denoted as

```
ClassName(parameterName: parameterType)
```

The function is denoted as

```
functionName(parameterName: parameterType): returnType
```

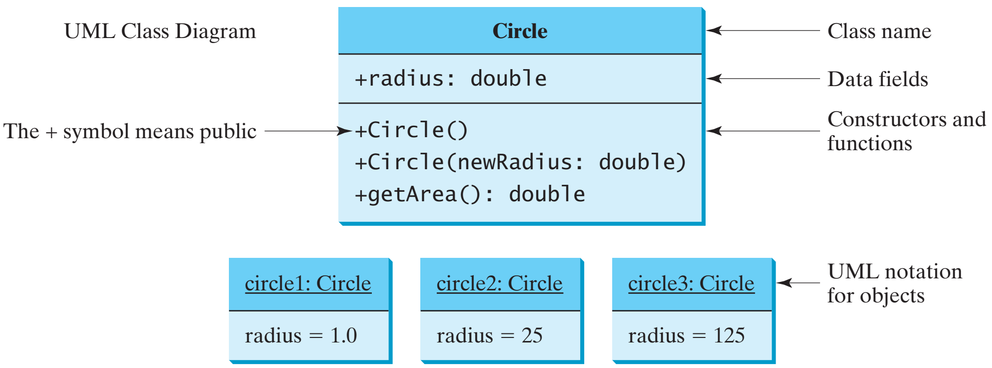

FIGURE 9.3 Classes and objects can be represented using UML notations.

9.3 Example: Defining Classes and Creating Objects

Key Point

Classes are definitions for objects and objects are created from classes.

Listing 9.1 is a program that demonstrates classes and objects. It constructs three circle objects with radius **1.0**, **25**, and **125** and displays the radius and area of each. Change the radius of the second object to **100** and display its new radius and area.

VideoNote
Use classes

LISTING 9.1 TestCircle.cpp

define class · data field · no-arg constructor · second constructor · function · don't omit · main function · creating object · creating object · creating object · accessing radius · invoking getArea · modify radius

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

class Circle
{
public:
  // The radius of this circle
  double radius;

  // Construct a default circle object
  Circle()
  {
    radius = 1;
  }

  // Construct a circle object
  Circle(double newRadius)
  {
    radius = newRadius;
  }

  // Return the area of this circle
  double getArea()
  {
    return radius * radius * 3.14159;
  }
};  // Must place a semicolon here

int main()
{
  Circle circle1(1.0);
  Circle circle2(25);
  Circle circle3(125);

  cout << "The area of the circle of radius "
    << circle1.radius << " is " << circle1.getArea() << endl;
  cout << "The area of the circle of radius "
    << circle2.radius << " is " << circle2.getArea() << endl;
  cout << "The area of the circle of radius "
    << circle3.radius << " is " << circle3.getArea() << endl;

  // Modify circle radius
  circle2.radius = 100;
  cout << "The area of the circle of radius "
    << circle2.radius << " is " << circle2.getArea() << endl;

  return 0;
}
```

```
The area of the circle of radius 1 is 3.14159
The area of the circle of radius 25 is 1963.49
The area of the circle of radius 125 is 49087.3
The area of the circle of radius 100 is 31415.9
```

ending class definition

public

private by default

The class is defined in lines 4–27. Don't forget that the semicolon (`;`) in line 27 is required.

The `public` keyword in line 6 denotes that all data fields, constructors, and functions can be accessed from the objects of the class. If you don't use the `public` keyword, the visibility is private by default. Private visibility will be introduced in Section 9.8.

The main function creates three objects named `circle1`, `circle2`, and `circle3` with radius `1.0`, `25`, and `125`, respectively (lines 31–33). These objects have different radii but the same functions. Therefore, you can compute their respective areas by using the `getArea()` function. The data fields can be accessed via the object using `circle1.radius`, `circle2.radius`, and `circle3.radius`, respectively. The functions are invoked using `circle1.getArea()`, `circle2.getArea()`, and `circle3.getArea()`, respectively.

These three objects are independent. The radius of `circle2` is changed to `100` in line 43. The object's new radius and area are displayed in lines 44–45.

As another example, consider TV sets. Each TV is an object with state (current channel, current volume level, power on or off) and behaviors (change channels, adjust volume, turn on/off). You can use a class to model TV sets. The UML diagram for the class is shown in Figure 9.4.

TV	
`channel: int`	The current channel (1 to 120) of this TV.
`volumeLevel: int`	The current volume level (1 to 7) of this TV.
`on: boolean`	Indicates whether this TV is on/off.
`+TV()`	Constructs a default TV object.
`+turnOn(): void`	Turns on this TV.
`+turnOff(): void`	Turns off this TV.
`+setChannel(newChannel: int): void`	Sets a new channel for this TV.
`+setVolume(newVolumeLevel: int): void`	Sets a new volume level for this TV.
`+channelUp(): void`	Increases the channel number by 1.
`+channelDown(): void`	Decreases the channel number by 1.
`+volumeUp(): void`	Increases the volume level by 1.
`+volumeDown(): void`	Decreases the volume level by 1.

FIGURE 9.4 The TV class models TV sets.

Listing 9.2 gives a program that defines the `TV` class and uses the `TV` class to create two objects.

LISTING 9.2 TV.cpp

define a class

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

class TV
{
public:
```

data fields

```
  int channel;
  int volumeLevel; // Default volume level is 1
  bool on; // By default TV is off

```

constructor

```
  TV()
  {
    channel = 1; // Default channel is 1
    volumeLevel = 1; // Default volume level is 1
    on = false; // By default TV is off
  }

```

turn on TV

```
  void turnOn()
  {
    on = true;
  }

```

turn off TV

```
  void turnOff()
  {
    on = false;
  }

```

set a new channel

```
  void setChannel(int newChannel)
  {
    if (on && newChannel >= 1 && newChannel <= 120)
      channel = newChannel;
  }

```

set a new volume

```
  void setVolume(int newVolumeLevel)
  {
    if (on && newVolumeLevel >= 1 && newVolumeLevel <= 7)
      volumeLevel = newVolumeLevel;
  }

```

increase channel

```
  void channelUp()
  {
    if (on && channel < 120)
      channel++;
  }

```

decrease channel

```
  void channelDown()
  {
    if (on && channel > 1)
      channel--;
  }

```

increase volume

```
  void volumeUp()
  {
    if (on && volumeLevel < 7)
      volumeLevel++;
  }

```

decrease volume

```
  void volumeDown()
  {
    if (on && volumeLevel > 1)
      volumeLevel--;
  }
};

```

main function

```
int main()
{
```

```
67    TV tv1;
68    tv1.turnOn();
69    tv1.setChannel(30);
70    tv1.setVolume(3);
71
72    TV tv2;
73    tv2.turnOn();
74    tv2.channelUp();
75    tv2.channelUp();
76    tv2.volumeUp();
77
78    cout << "tv1's channel is " << tv1.channel
79      << " and volume level is " << tv1.volumeLevel << endl;
80    cout << "tv2's channel is " << tv2.channel
81      << " and volume level is " << tv2.volumeLevel << endl;
82
83    return 0;
84  }
```

create a TV
turn on
set a new channel
set a new volume
create a TV
turn on
increase channel
increase volume
display state

```
tv1's channel is 30 and volume level is 3
tv2's channel is 3 and volume level is 2
```

Note that the channel and volume level are not changed if the TV is not on. Before changing a channel or volume level, the current values are checked to ensure that the channel and volume level are within the correct range.

The program creates two objects in lines 67 and 72, and invokes the functions on the objects to perform actions for setting channels and volume levels and for increasing channels and volumes. The program displays the state of the objects in lines 78–81. The functions are invoked using a syntax such as `tv1.turnOn()` (line 68). The data fields are accessed using a syntax such as `tv1.channel` (line 78).

These examples have given you a glimpse of classes and objects. You may have many questions about constructors and objects, accessing data fields and invoking objects' functions. The sections that follow discuss these issues in detail.

9.4 Constructors

A constructor is invoked to create an object.

Key Point

Constructors are a special kind of function, with three peculiarities:

- Constructors must have the same name as the class itself. (constructor's name)
- Constructors do not have a return type—not even `void`. (no return type)
- Constructors are invoked when an object is created. Constructors play the role of initializing objects. (invoke constructor)

The constructor has exactly the same name as the defining class. Like regular functions, constructors can be overloaded (i.e., multiple constructors with the same name but different signatures), making it easy to construct objects with different sets of data values.

constructor overloading

It is a common mistake to put the `void` keyword in front of a constructor. For example,

no `void`

```
void Circle()
{
}
```

Most C++ compilers will report an error, but some will treat this as a regular function, not as a constructor.

initialize data field

Constructors are for initializing data fields. The data field **radius** does not have an initial value, so it must be initialized in the constructor (lines 13 and 19 in Listing 9.1). Note that a variable (local or global) can be declared and initialized in one statement, but as a class member, a data field cannot be initialized when it is declared. For example, it would be wrong to replace line 8 in Listing 9.1 by

```
double radius = 5; // Wrong for data field declaration
```

no-arg constructor

A class normally provides a constructor without arguments (e.g., **Circle()**). Such constructor is called a *no-arg* or *no-argument constructor*.

default constructor

A class may be defined without constructors. In this case, a no-arg constructor with an empty body is implicitly defined in the class. Called a *default constructor*, it is provided automatically *only if no constructors are explicitly defined in the class*.

constructor initializer list

Data fields may be initialized in the constructor using an initializer list in the following syntax:

```
ClassName(parameterList)
  : datafield1(value1), datafield2(value2) // Initializer list
{
  // Additional statements if needed
}
```

The initializer list initializes **datafield1** with **value1** and **datafield2** with **value2**. For example,

```
Circle::Circle()
  : radius(1)
{
}
```

(a)

same as

```
Circle::Circle()
{
  radius = 1;
}
```

(b)

Constructor in (b), which does not use an initializer list, is actually more intuitive than the one in (a). However, using an initializer list is necessary to initialize object data fields that don't have a no-arg constructor. This is an advanced topic covered in Supplement IV.E on the Companion Website.

9.5 Constructing and Using Objects

Key Point

*An object's data and functions can be accessed through the dot (**.**) operator via the object's name.*

construct objects

A constructor is invoked when an object is created. The syntax to create an object using the no-arg constructor is

invoke no-arg constructor

```
ClassName objectName;
```

For example, the following declaration creates an object named **circle1** by invoking the **Circle** class's no-arg constructor.

```
Circle circle1;
```

The syntax to create an object using a constructor with arguments is

construct with args

```
ClassName objectName(arguments);
```

For example, the following declaration creates an object named **circle2** by invoking the **Circle** class's constructor with a specified radius **5.5**.

```
Circle circle2(5.5);
```

In OOP term, an object's member refers to its data fields and functions. Newly created objects are allocated in the memory. After an object is created, its data can be accessed and its functions invoked using the *dot operator* (**.**), also known as the *object member access operator*:

dot operator
member access operator

- **objectName.dataField** references a data field in the object.
- **objectName.function(arguments)** invokes a function on the object.

For example, **circle1.radius** references the radius in **circle1**, and **circle1.getArea()** invokes the **getArea** function on **circle1**. Functions are invoked as operations on objects.

The data field **radius** is referred to as an *instance member variable* or simply *instance variable*, because it is dependent on a specific instance. For the same reason, the function **getArea** is referred to as an *instance member function* or *instance function*, because you can invoke it only on a specific instance. The object on which an instance function is invoked is called a *calling object*.

instance variable
member function
instance function
calling object

Note

When you define a custom class, capitalize the first letter of each word in a class name—for example, the class names **Circle**, **Rectangle**, and **Desk**. The class names in the C++ library are named in lowercase. The objects are named like variables.

class naming convention
object naming convention

The following points on classes and objects are worth noting:

- You can use primitive data types to define variables. You can also use class names to declare object names. In this sense, a class is also a data type.

class is a type

- In C++, you can use the assignment operator **=** to copy the contents from one object to the other. By default, each data field of one object is copied to its counterpart in the other object. For example,

memberwise copy

```
circle2 = circle1;
```

copies the **radius** in **circle1** to **circle2**. After the copy, **circle1** and **circle2** are still two different objects but have the same radius.

- Object names are like array names. Once an object name is declared, it represents an object. It cannot be reassigned to represent another object. In this sense, an object name is a constant, though the contents of the object may change. Memberwise copy can change an object's contents but not its name.

constant object name

- An object contains data and may invoke functions. This may lead you to think that an object is quite large. It isn't, though. Data are physically stored in an object, but functions are not. Since functions are shared by all objects of the same class, the compiler creates just one copy for sharing. You can find out the actual size of an object using the **sizeof** function. For example, the following code displays the size of objects **circle1** and **circle2**. Their size is **8**, since the data field radius is **double**, which takes **8** bytes.

object size

```
Circle circle1;
Circle circle2(5.0);

cout << sizeof(circle1) << endl;
cout << sizeof(circle2) << endl;
```

anonymous objects

Usually you create a named object and later access its members through its name. Occasionally you may create an object and use it only once. In this case, you don't have to name it. Such objects are called *anonymous objects*.

The syntax to create an anonymous object using the no-arg constructor is

```
ClassName()
```

The syntax to create an anonymous object using the constructor with arguments is

```
ClassName(arguments)
```

For example,

```
circle1 = Circle();
```

creates a `Circle` object using the no-arg constructor and copies its contents to `circle1`.

```
circle1 = Circle(5);
```

creates a `Circle` object with radius `5` and copies its contents to `circle1`.

For example, the following code creates `Circle` objects and invokes their `getArea()` function.

```
cout << "Area is " << Circle().getArea() << endl;
cout << "Area is " << Circle(5).getArea() << endl;
```

As you see from these examples, you may create an anonymous object if it will not be referenced later.

no-arg constructor

Caution

Please note that in C++, to create an anonymous object using the *no-arg constructor*, you have to add parentheses after the constructor name (e.g., `Circle()`). To create a named object using the no-arg constructor, you cannot use the parentheses after the constructor name (e.g., you use `Circle circle1` rather than `Circle circle1()`). This is the required syntax, which you just have to accept.

Check Point

9.1 Describe the relationship between an object and its defining class. How do you define a class? How do you declare and create an object?

9.2 What are the differences between constructors and functions?

9.3 How do you create an object using a no-arg constructor? How do you create an object using a constructor with arguments?

9.4 Once an object name is declared, can it be reassigned to reference another object?

9.5 Assuming that the `Circle` class is defined as in Listing 9.1, show the printout of the following code:

```
Circle c1(5);
Circle c2(6);
c1 = c2;
cout << c1.radius << " " << c2.radius << endl;
```

9.6 What is wrong in the following code? (Use the `Circle` class defined in Listing 9.1, TestCircle.cpp.)

```
int main()
{
  Circle c1();
  cout << c1.getRadius() << endl;

  return 0;
}
```

(a)

```
int main()
{
  Circle c1(5);
  Circle c1(6);

  return 0;
}
```

(b)

9.7 What is wrong in the following code?

```
class Circle
{
public:
  Circle()
  {
  }
  double radius = 1;
};
```

9.8 Which of the following statements is correct?

```
Circle c;

Circle c();
```

9.9 Suppose the following two are independent statements. Are they correct?

```
Circle c;

Circle c = Circle();
```

VideoNote

Separate class definition

9.6 Separating Class Definition from Implementation

Key Point

Separating class definition from class implementation makes the class easy to maintain.

C++ allows you to separate class definition from implementation. The class definition describes the *contract* of the class and the class implementation carries out the contract. The class definition simply lists all the data fields, constructor prototypes, and function prototypes. The class implementation implements the constructors and functions. The class definition and implementation may be in two separate files. Both files should have the same name but different extension names. The class definition file has an extension name `.h` (h means header) and the class implementation file an extension name `.cpp`.

Listings 9.3 and 9.4 present the `Circle` class definition and implementation.

LISTING 9.3 Circle.h

```
class Circle
{
public:
  // The radius of this circle
  double radius;                                    // data field

  // Construct a default circle object
  Circle();                                         // no-arg constructor
```

second constructor

function prototype

semicolon required

```

  // Construct a circle object
  Circle(double);

  // Return the area of this circle
  double getArea();
};
```

don't omit semicolon

Caution

It is a common mistake to omit the semicolon (;) at the end of the class definition.

LISTING 9.4 Circle.cpp

include class definition

implement constructor

implement constructor

implement function

```
#include "Circle.h"

// Construct a default circle object
Circle::Circle()
{
  radius = 1;
}

// Construct a circle object
Circle::Circle(double newRadius)
{
  radius = newRadius;
}

// Return the area of this circle
double Circle::getArea()
{
  return radius * radius * 3.14159;
}
```

binary scope resolution operator

The `::` symbol, known as the *binary scope resolution operator*, specifies the scope of a class member in a class.

Here, `Circle::` preceding each constructor and function in the `Circle` class tells the compiler that these constructors and functions are defined in the `Circle` class.

client

Listing 9.5 is a program that uses the `Circle` class. Such a program that uses the class is often referred to as a *client* of the class.

LISTING 9.5 TestCircleWithHeader.cpp

include class definition

construct circle

construct circle

set a new radius

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

int main()
{
  Circle circle1;
  Circle circle2(5.0);

  cout << "The area of the circle of radius "
    << circle1.radius << " is " << circle1.getArea() << endl;
  cout << "The area of the circle of radius "
    << circle2.radius << " is " << circle2.getArea() << endl;

  // Modify circle radius
  circle2.radius = 100;
```

```
    cout << "The area of the circle of radius "
      << circle2.radius << " is " << circle2.getArea() << endl;

    return 0;
  }
```

```
The area of the circle of radius 1 is 3.14159
The area of the circle of radius 5 is 78.5397
The area of the circle of radius 100 is 31415.9
```

why separation?

There are at least two benefits for separating a class definition from implementation.

1. It hides implementation from definition. You can feel free to change the implementation. The client program that uses the class does not need to change as long as the definition is not changed.
2. As a software vendor, you can just provide the customer with the header file and class object code without revealing the source code for implementing the class. This protects the software vendor's intellectual property.

compile from command line

Note

To compile a main program from the command line, you need to add all its supporting files in the command. For example, to compile TestCircleWithDefinition.cpp using a GNU C++ compiler, the command is

```
g++ Circle.h Circle.cpp TestCircleWithHeader.cpp -o Main
```

compile from IDE

Note

If the main program uses other programs, all of these program source files must be present in the project pane in the IDE. Otherwise, you may get linking errors. For example, to run `TestCircleWithHeader.cpp`, you need to place `TestCircleWithHeader.cpp`, `Circle.cpp`, and `Circle.h` in the project pane in Visual C++, as shown in Figure 9.5.

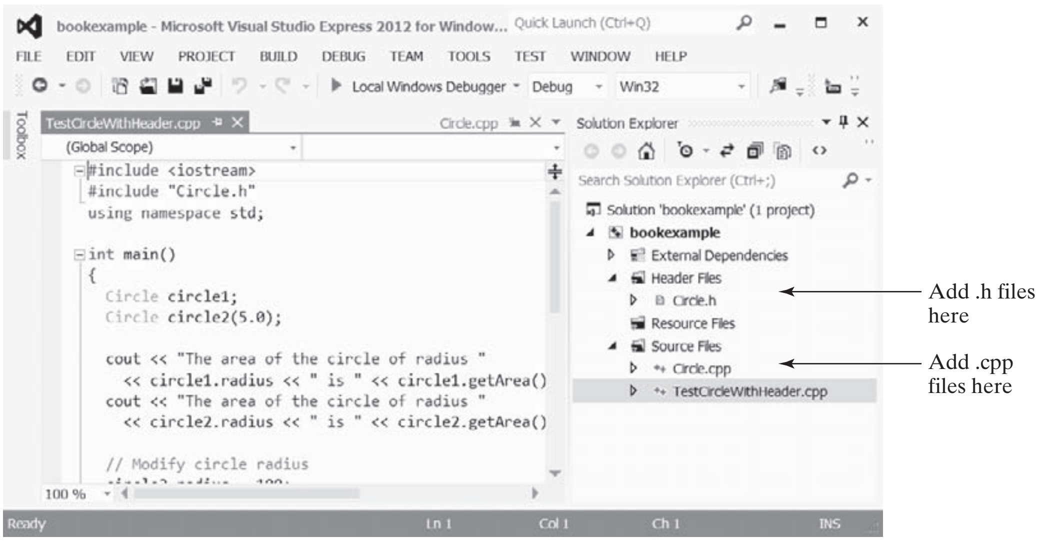

FIGURE 9.5 For the program to run, you need to place all dependent files in the project pane.

Check Point

9.10 How do you separate class definition from implementation?

9.11 What is the output of the following code? (Use the `Circle` class defined in Listing 9.3, Circle.h.)

```
int main()
{
  Circle c1;
  Circle c2(6);
  c1 = c2;
  cout << c1.getArea() << endl;

  return 0;
}
```

(a)

```
int main()
{
  cout << Circle(8).getArea()
    << endl;

  return 0;
}
```

(b)

9.7 Preventing Multiple Inclusions

Key Point

Inclusion guard prevents header files to be included multiple times.

It is a common mistake to include, inadvertently, the same header file in a program multiple times. Suppose `Head.h` includes `Circle.h` and TestHead.cpp includes both `Head.h` and `Circle.h`, as shown in Listings 9.6 and 9.7.

Listing 9.6 Head.h

include `Circle.h`

```
1 #include "Circle.h"
2 // Other code in Head.h omitted
```

Listing 9.7 TestHead.cpp

include `Circle.h`
include `Head.h`

```
1 #include "Circle.h"
2 #include "Head.h"
3
4 int main()
5 {
6   // Other code in TestHead.cpp omitted
7 }
```

If you compile TestHead.cpp, you will get a compile error indicating that there are multiple definitions for `Circle`. What is wrong here? Recall that the C++ preprocessor inserts the contents of the header file at the position where the header is included. `Circle.h` is included in line 1. Since the header file for `Circle` is also included in `Head.h` (see line 1 in Listing 9.6), the preprocessor will add the definition for the `Circle` class another time as result of including `Head.h` in TestHead.cpp, which causes the multiple-inclusion errors.

The C++ `#ifndef` directive along with the `#define` directive can be used to prevent a header file from being included multiple times. This is known as *inclusion guard*. To make this work, you have to add three lines to the header file. The three lines are highlighted in Listing 9.8.

inclusion guard

Listing 9.8 CircleWithInclusionGuard.h

is symbol defined?
define symbol

```
#ifndef CIRCLE_H
#define CIRCLE_H

class Circle
{
public:
```

```
 7     // The radius of this circle
 8     double radius;
 9
10     // Construct a default circle object
11     Circle();
12
13     // Construct a circle object
14     Circle(double);
15
16     // Return the area of this circle
17     double getArea();
18   };
19
20   #endif
```

end of #ifndef

Recall that the statements preceded by the pound sign (#) are preprocessor directives. They are interpreted by the C++ preprocessor. The preprocessor directive **#ifndef** stands for "*if n*ot *def*ined." Line 1 tests whether the symbol **CIRCLE_H** is already defined. If not, define the symbol in line 2 using the **#define** directive and the rest of the header file is included; otherwise, the rest of the header file is skipped. The **#endif** directive is needed to indicate the end of header file.

To avoid multiple-inclusion errors, define a class using the following template and convention for naming the symbol:

```
#ifndef ClassName_H
#define ClassName_H

A class header for the class named ClassName

#endif
```

If you replace Circle.h by CircleWithInclusionGuard.h in Listings 9.6 and 9.7, the program will not have the multiple-inclusion error.

Check Point

9.12 What might cause multiple-inclusion errors? How do you prevent multiple inclusions of header files?

9.13 What is the **#define** directive for?

9.8 Inline Functions in Classes

Key Point

You can define short functions as inline functions to improve performance.

Section 6.10, "Inline Functions," introduced how to improve function efficiency using inline functions. When a function is implemented inside a class definition, it automatically becomes an inline function. This is also known as *inline definition*. For example, in the following definition for class **A**, the constructor and function **f1** are automatically inline functions, but function **f2** is not.

inline definition

```
class A
{
public:
  A()
  {
    // Do something;
  }

  double f1()
  {
    // Return a number
  }
```

```
  double f2();
};
```

There is another way to define inline functions for classes. You may define inline functions in the class's implementation file. For example, to define function `f2` as an inline function, precede the inline keyword in the function header as follows:

```
// Implement function as inline
inline double A::f2()
{
  // Return a number
}
```

As noted in Section 6.10, short functions are good candidates for inline functions, but long functions are not.

Check Point

9.14 How do you implement all functions inline in Listing 9.4, Circle.cpp?

9.9 Data Field Encapsulation

Key Point

Making data fields private protects data and makes the class easy to maintain.

The data fields `radius` in the `Circle` class in Listing 9.1 can be modified directly (e.g., `circle1.radius = 5`). This is not a good practice—for two reasons:

- First, data may be tampered with.
- Second, it makes the class difficult to maintain and vulnerable to bugs. Suppose you want to modify the `Circle` class to ensure that the radius is nonnegative after other programs have already used the class. You have to change not only the `Circle` class, but also the programs that use the `Circle` class. This is because the clients may have modified the radius directly (e.g., `myCircle.radius = -5`).

data field encapsulation
private

To prevent direct modifications of properties, you should declare the data field private, using the `private` keyword. This is known as *data field encapsulation*. Making the radius data field private in the `Circle` class, you can define the class as follows:

```
class Circle
{
public:
  Circle();
  Circle(double);
  double getArea();

private:
  double radius;
};
```

A private data field cannot be accessed by an object through a direct reference outside the class that defines the private field. But often a client needs to retrieve and/or modify a data field. To make a private data field accessible, provide a *get* function to return the field's value. To enable a private data field to be updated, provide a *set* function to set a new value.

accessor
mutator

Note

Colloquially, a `get` function is referred to as an *accessor*, and a `set` function is referred to as a *mutator*.

A **get** function has the following signature:

```
returnType getPropertyName()
```

If the **returnType** is **bool**, by convention the **get** function should be defined as follows:

bool accessor

```
bool isPropertyName()
```

A **set** function has the following signature:

```
void setPropertyName(dataType propertyValue)
```

Let us create a new circle class with a private data field radius and its associated accessor and mutator functions. The class diagram is shown in Figure 9.6. The new circle class is defined in Listing 9.9.

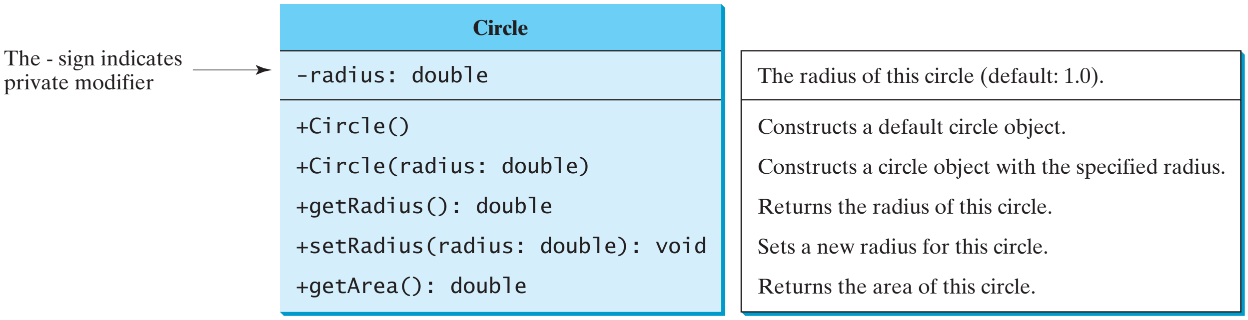

FIGURE 9.6 The **Circle** class encapsulates circle properties and provides get/set and other functions.

LISTING 9.9 CircleWithPrivateDataFields.h

public
access function
mutator function
private

```
#ifndef CIRCLE_H
#define CIRCLE_H

class Circle
{
public:
  Circle();
  Circle(double);
  double getArea();
  double getRadius();
  void setRadius(double);

private:
  double radius;
};

#endif
```

Listing 9.10 implements the class contract specified in the header file in Listing 9.9.

LISTING 9.10 CircleWithPrivateDataFields.cpp

include header file
constructor

```
#include "CircleWithPrivateDataFields.h"

// Construct a default circle object
Circle::Circle()
```

constructor

get area

get radius

set radius

```
5  {
6    radius = 1;
7  }
8
9  // Construct a circle object
10 Circle::Circle(double newRadius)
11 {
12   radius = newRadius;
13 }
14
15 // Return the area of this circle
16 double Circle::getArea()
17 {
18   return radius * radius * 3.14159;
19 }
20
21 // Return the radius of this circle
22 double Circle::getRadius()
23 {
24   return radius;
25 }
26
27 // Set a new radius
28 void Circle::setRadius(double newRadius)
29 {
30   radius = (newRadius >= 0) ? newRadius : 0;
31 }
```

The **getRadius()** function (lines 22–25) returns the radius, and the **setRadius (newRadius)** function (line 28–31) sets a new radius into the object. If the new radius is negative, **0** is set to the radius in the object. Since these functions are the only ways to read and modify radius, you have total control over how the **radius** property is accessed. If you have to change the functions' implementation, you need not change the client programs. This makes the class easy to maintain.

Listing 9.11 is a client program that uses the **Circle** class to create a **Circle** object and modifies the radius using the **setRadius** function.

LISTING 9.11 TestCircleWithPrivateDataFields.cpp

include header file

construct object

construct object

get radius

set radius

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

int main()
{
  Circle circle1;
  Circle circle2(5.0);

  cout << "The area of the circle of radius "
    << circle1.getRadius() << " is " << circle1.getArea() << endl;
  cout << "The area of the circle of radius "
    << circle2.getRadius() << " is " << circle2.getArea() << endl;

  // Modify circle radius
  circle2.setRadius(100);
  cout << "The area of the circle of radius "
    << circle2.getRadius() << " is " << circle2.getArea() << endl;

  return 0;
}
```

```
The area of the circle of radius 1 is 3.14159
The area of the circle of radius 5 is 78.5397
The area of the circle of radius 100 is 31415.9
```

The data field **radius** is declared private. Private data can be accessed only within their defining class. You cannot use **circle1.radius** in the client program. A compile error would occur if you attempted to access private data from a client.

Tip

To prevent data from being tampered with and to make the class easy to maintain, the data fields in this book will be private.

Check Point

9.15 What is wrong in the following code? (Use the **Circle** class defined in Listing 9.9, CircleWithPrivateDataFields.h.)

```
Circle c;
cout << c.radius << endl;
```

9.16 What is an accessor function? What is a mutator function? What are the naming conventions for such functions?

9.17 What are the benefits of data field encapsulation?

9.10 The Scope of Variables

Key Point

The scope of instance and static variables is the entire class, regardless of where the variables are declared.

Chapter 6 discussed the scope of global variables, local variables, and static local variables. Global variables are declared outside all functions and are accessible to all functions in its scope. The scope of a global variable starts from its declaration and continues to the end of the program. Local variables are defined inside functions. The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. Static local variables are permanently stored in the program so they can be used in the next call of the function.

The data fields are declared as variables and are accessible to all constructors and functions in the class. Data fields and functions can be in any order in a class. For example, all the following declarations are the same:

```
class Circle
{
public:
  Circle();
  Circle(double);
  double getArea();
  double getRadius();
  void setRadius(double);

private:
  double radius;
};
```

(a)

```
class Circle
{
public:
  Circle();
  Circle(double);

private:
  double radius;

public:
  double getArea();
  double getRadius();
  void setRadius(double);
};
```

(b)

```
class Circle
{
private:
  double radius;

public:
  double getArea();
  double getRadius();
  void setRadius(double);

public:
  Circle();
  Circle(double);
};
```

(c)

public first

Tip

Though the class members can be in any order, the common style in C++ is to place public members first and then private members.

This section discusses the scope rules of all the variables in the context of a class.

You can declare a variable for data field only once, but you can declare the same variable name in a function many times in different functions.

Local variables are declared and used inside a function locally. If a local variable has the same name as a data field, the local variable takes precedence, and the data field with the same name is hidden. For example, in the program in Listing 9.12, **x** is defined as a data field and as a local variable in the function.

Listing 9.12 HideDataField.cpp

data field x
data field y
no-arg constructor
local variable
create object
invoke function

```
1  #include <iostream>
2  using namespace std;
3
4  class Foo
5  {
6  public:
7    int x; // Data field
8    int y; // Data field
9
10   Foo()
11   {
12     x = 10;
13     y = 10;
14   }
15
16   void p()
17   {
18     int x = 20; // Local variable
19     cout << "x is " << x << endl;
20     cout << "y is " << y << endl;
21   }
22 };
23
24 int main()
25 {
26   Foo foo;
27   foo.p();
28
29   return 0;
30 }
```

```
x is 20
y is 10
```

Why is the printout **20** for **x** and **10** for **y**? Here is why:

- **x** is declared as a data field in the **Foo** class, but is also defined as a local variable in the function **p()** with an initial value of **20**. The latter **x** is displayed to the console in line 19.
- **y** is declared as a data field, so it is accessible inside function **p()**.

Tip

As demonstrated in the example, it is easy to make mistakes. To avoid confusion, do not declare the same variable name twice in a class, except for function parameters.

Check Point

9.18 Can data fields and functions be placed in any order in a class?

9.11 Class Abstraction and Encapsulation

VideoNote
The Loan class

Key Point

Class abstraction is the separation of class implementation from the use of a class. The details of implementation are encapsulated and hidden from the user. This is known as class encapsulation.

In Chapter 6 you learned about function abstraction and used it in stepwise program development. C++ provides many levels of abstraction. *Class abstraction* is the separation of class implementation from the use of a class. The creator of a class provides a description of the class and lets the user know how it can be used. The collection of functions and fields that are accessible from outside the class, together with the description of how these members are expected to behave, serves as the *class's contract*. As shown in Figure 9.7, the user of the class does not need to know how the class is implemented. The details of implementation are encapsulated and hidden from the user. This is known as *class encapsulation*. For example, you can create a **Circle** object and find the area of the circle without knowing how the area is computed.

class abstraction

class's contract

class encapsulation

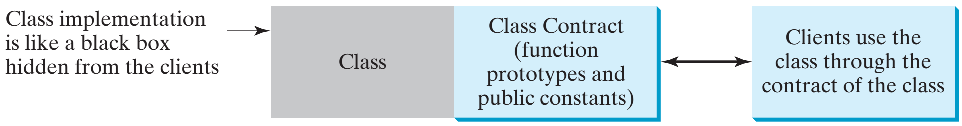

FIGURE 9.7 Class abstraction separates class implementation from the use of the class.

Class abstraction and encapsulation are two sides of the same coin. Many real-life examples illustrate the concept of class abstraction. Consider, for instance, building a computer system. Your personal computer is made up of many components, such as a CPU, CD-ROM, floppy disk, motherboard, fan, and so on. Each component can be viewed as an object that has properties and functions. To get the components to work together, all you need to know is how each component is used and how it interacts with the others. You don't need to know how it works internally. The internal implementation is encapsulated and hidden from you. You can build a computer without knowing how a component is implemented.

The computer-system analogy precisely mirrors the object-oriented approach. Each component can be viewed as an object of the class for the component. For example, you might have a class that models all kinds of fans for use in a computer, with properties like fan size and speed, functions like start, stop, and so on. A specific fan is an instance of this class with specific property values.

As another example, consider getting a loan. A specific loan can be viewed as an object of a **Loan** class. Interest rate, loan amount, and loan period are its data properties, and computing monthly payment and total payment are its functions. When you buy a car, a loan object is created by instantiating the class with your loan interest rate, loan amount, and loan period. You can then use the functions to find the monthly payment and total payment of your loan. As a user of the **Loan** class, you don't need to know how these functions are implemented.

Let us use the **Loan** class as an example to demonstrate the creation and use of classes. **Loan** has the data fields **annualInterestRate**, **numberOfYears**, and **loanAmount**, and the functions **getAnnualInterestRate**, **getNumberOfYears**, **getLoanAmount**, **setAnnualInterestRate**, **setNumberOfYears**, **setLoanAmount**, **getMonthlyPayment**, and **getTotalPayment**, as shown in Figure 9.8.

Loan	
-annualInterestRate: double	The annual interest rate of the loan (default: 2.5).
-numberOfYears: int	The number of years for the loan (default: 1)
-loanAmount: double	The loan amount (default: 1000).
+Loan()	Constructs a default loan object.
+Loan(rate: double,years: int, amount: double)	Constructs a loan with specified interest rate, years, and loan amount.
+getAnnualInterestRate(): double	Returns the annual interest rate of this loan.
+getNumberOfYears(): int	Returns the number of the years of this loan.
+getLoanAmount(): double	Returns the amount of this loan.
+setAnnualInterestRate(rate: double): void	Sets a new annual interest rate to this loan.
+setNumberOfYears(years: int): void	Sets a new number of years to this loan.
+setLoanAmount(amount: double): void	Sets a new amount to this loan.
+getMonthlyPayment(): double	Returns the monthly payment of this loan.
+getTotalPayment(): double	Returns the total payment of this loan.

FIGURE 9.8 The **Loan** class models the properties and behaviors of loans.

The UML diagram in Figure 9.8 serves as the contract for the **Loan** class. Throughout the book, you will play the role of both class user and class developer. The user can use the class without knowing how the class is implemented. Assume that the **Loan** class is available, with the header file, as shown in Listing 9.13. Let us begin by writing a test program that uses the **Loan** class, in Listing 9.14.

LISTING 9.13 Loan.h

public functions

private fields

```
#ifndef LOAN_H
#define LOAN_H

class Loan
{
public:
  Loan();
  Loan(double rate, int years, double amount);
  double getAnnualInterestRate();
  int getNumberOfYears();
  double getLoanAmount();
  void setAnnualInterestRate(double rate);
  void setNumberOfYears(int years);
  void setLoanAmount(double amount);
  double getMonthlyPayment();
  double getTotalPayment();

private:
  double annualInterestRate;
```

```
  int numberOfYears;
  double loanAmount;
};

#endif
```

LISTING 9.14 TestLoanClass.cpp

```
1  #include <iostream>
2  #include <iomanip>
3  #include "Loan.h"
4  using namespace std;
5
6  int main()
7  {
8    // Enter annual interest rate
9    cout << "Enter yearly interest rate, for example 8.25: ";
10   double annualInterestRate;
11   cin >> annualInterestRate;
12
13   // Enter number of years
14   cout << "Enter number of years as an integer, for example 5: ";
15   int numberOfYears;
16   cin >> numberOfYears;
17
18   // Enter loan amount
19   cout << "Enter loan amount, for example 120000.95: ";
20   double loanAmount;
21   cin >> loanAmount;
22
23   // Create Loan object
24   Loan loan(annualInterestRate, numberOfYears, loanAmount);
25
26   // Display results
27   cout << fixed << setprecision(2);
28   cout << "The monthly payment is "
29     << loan.getMonthlyPayment() << endl;
30   cout << "The total payment is " << loan.getTotalPayment() << endl;
31
32   return 0;
33 }
34
```

include Loan header

input number of years

input loan amount

create Loan object

monthly payment

total payment

The **main** function reads interest rate, payment period (in years), and loan amount (lines 8–21), creates a **Loan** object (line 24), and then obtains the monthly payment (line 29) and total payment (line 30) using the instance functions in the **Loan** class.

The **Loan** class can be implemented as in Listing 9.15.

LISTING 9.15 Loan.cpp

```
#include "Loan.h"
#include <cmath>
using namespace std;

Loan::Loan()
{
  annualInterestRate = 9.5;
  numberOfYears = 30;
  loanAmount = 100000;
}
```

no-arg constructor

constructor

```
Loan::Loan(double rate, int years, double amount)
{
  annualInterestRate = rate;
  numberOfYears = years;
  loanAmount = amount;
}
```

accessor function

```
double Loan::getAnnualInterestRate()
{
  return annualInterestRate;
}
```

accessor function

```
int Loan::getNumberOfYears()
{
  return numberOfYears;
}
```

accessor function

```
double Loan::getLoanAmount()
{
  return loanAmount;
}
```

mutator function

```
void Loan::setAnnualInterestRate(double rate)
{
  annualInterestRate = rate;
}
```

mutator function

```
void Loan::setNumberOfYears(int years)
{
  numberOfYears = years;
}
```

mutator function

```
void Loan::setLoanAmount(double amount)
{
  loanAmount = amount;
}
```

get monthly payment

```
double Loan::getMonthlyPayment()
{
  double monthlyInterestRate = annualInterestRate / 1200;
  return loanAmount * monthlyInterestRate / (1 -
    (pow(1 / (1 + monthlyInterestRate), numberOfYears * 12)));
}
```

get total payment

```
double Loan::getTotalPayment()
{
  return getMonthlyPayment() * numberOfYears * 12;
}
```

From a class developer's perspective, a class is designed for use by many different customers. In order to be useful in a wide range of applications, a class should provide a variety of ways for customization through constructors, properties, and functions.

The `Loan` class contains two constructors, three *get* functions, three *set* functions, and the functions for finding monthly payment and total payment. You can construct a `Loan` object by using the no-arg constructor or the one with three parameters: annual interest rate, number of years, and loan amount. The three *get* functions, `getAnnualInterest`, `getNumberOfYears`, and `getLoanAmount`, return annual interest rate, payment years, and loan amount, respectively.

Important Pedagogical Tip

The UML diagram for the **Loan** class is shown in Figure 9.8. Students should begin by writing a test program that uses the **Loan** class even though they don't know how the **Loan** class is implemented. This has three benefits:

- It demonstrates that developing a class and using a class are two separate tasks.
- It enables you to skip the complex implementation of certain classes without interrupting the sequence of the book.
- It is easier to learn how to implement a class if you are familiar with the class through using it.

For all the examples from now on, you may first create an object from the class and try to use its functions before turning your attention to its implementation.

9.19 What is the output of the following code? (Use the **Loan** class defined in Listing 9.13, Loan.h.)

Check Point

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

class A
{
public:
  Loan loan;
  int i;
};

int main()
{
  A a;
  cout << a.loan.getLoanAmount() << endl;
  cout << a.i << endl;

  return 0;
}
```

Key Terms

accessor 376
anonymous object 370
binary scope resolution operator (`::`) 372
calling object 369
class 362
class abstraction 381
class encapsulation 381
client 372
constructor 362
constructor initializer list 382
contract 362
data field 362
data field encapsulation 376
default constructor 368
dot operator (`.`) 369
inclusion guard 374
inline definition 375
instance 362
instance function 369
instance variable 369
instantiation 362
member function 369
member access operator 369
mutator 376
no-arg constructor 368
object 362
object-oriented programming (OOP) 362
property 362
`private` 376
`public` 365
state 362
UML class diagram 363

Chapter Summary

1. A class is a blueprint for objects.

2. A class defines the data fields for storing the properties of objects and provides constructors for creating objects and functions for manipulating them.

3. Constructors must have the same name as the class itself.

4. A non-arg constructor is a constructor that does not have arguments.

5. A class is also a data type. You can use it to declare and create objects.

6. An object is an instance of a class. You use the dot (`.`) operator to access members of that object through its name.

7. The *state* of an object is represented by *data fields* (also known as *properties*) with their current values.

8. The *behavior* of an object is defined by a set of functions.

9. The data fields do not have initial values. They must be initialized in constructors.

10. You can separate class definition from class implementation by defining class in a header file and class implementation in a separate file.

11. The C++ `#ifndef` directive, called *inclusion guard*, can be used to prevent a header file from being included multiple times.

12. When a function is implemented inside a class definition, it automatically becomes an inline function.

13. Visibility keywords specify how the class, function, and data are accessed.

14. A `public` function or data is accessible to all clients.

15. A `private` function or data is accessible only inside the class.

16. You can provide a *get* function or a *set* function to enable clients to see or modify the data.

17. Colloquially, a *get* function is referred to as a *getter* (or *accessor*), and a *set* function is referred to as a *setter* (or *mutator*).

18. A *get* function has the signature

    ```
    returnType getPropertyName()
    ```

19. If the `returnType` is `bool`, the *get* function should be defined as

    ```
    bool isPropertyName().
    ```

20. A *set* function has the signature

    ```
    void setPropertyName(dataType propertyValue)
    ```