CHAPTER
9
Objects and Classes
Objectives

- To understand objects and classes, and use classes to model objects (§9.2).
- To use UML graphical notations to describe classes and objects (§9.2).
- To understand the role of constructors when creating objects (§9.3).
- To learn how to declare a class and how to create an object of a class (§9.4).
- To know how to separate a class interface from a class implementation (§9.5).
- To access object members using pointers (§9.6).
- To create objects using the new operator on the heap (§9.7).
- To process strings using the C++ string class (§9.8).
- To declare private data fields with appropriate get and set functions for data field encapsulation to make classes easy to maintain (§9.9).
- To understand the scope of data fields (§9.10).
- To reference hidden data field using the this pointer (§9.11).
- To develop functions with object arguments (§9.12).
- To store and process objects in arrays (§9.13).
- To apply class abstraction to develop software (§§9.14–9.15).
9.1 Introduction
Programming in procedural languages like C, Pascal, BASIC, Ada, and COBOL involves choosing data structures, designing algorithms, and translating algorithms into code. An object-oriented language like C++ combines the power of procedural languages with an added dimension that provides more flexibility, modularity, clarity, and reusability through abstraction, encapsulation, inheritance, and polymorphism.

In procedural programming, data and operations on the data are separate, and this methodology requires sending data to procedures and functions. Object-oriented programming places data and the operations that pertain to them within a single entity called an object; this approach solves many of the problems inherent in procedural programming. The object-oriented programming approach organizes programs in a way that mirrors the real world, in which all objects are associated with both attributes and activities. Using objects improves software reusability and makes programs easier to develop and easier to maintain. Object-oriented programming involves thinking in terms of objects; a program can be viewed as a collection of cooperating objects.

This chapter introduces declaring classes, creating objects, manipulating objects, and making objects work together.

9.2 Defining Classes for Objects
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 behaviors.

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

- The behavior of an object is defined by a set of functions. Invoking a function on an object is to ask the object to perform a task.

A circle object, for example, has a data field, radius, which is the property that characterizes a circle. One behavior of a circle is that its area can be computed using the function getArea().
Objects of the same type are defined using a common class. A class is a template or a blueprint that defines what an object’s data 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.

![Class Name: Circle](image)

Data Fields:
- radius is ______

Methods:
- getArea

Figure 9.1
A class is a template for creating objects.

*Side Remark: class*
*Side Remark: data field*
*Side Remark: function*
*Side Remark: constructor*

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. A constructor can perform any action, but constructors 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.
A class is a construct that defines objects of the same type.

<Side Remark: class diagram>

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

\[ \text{dataField}_\text{Name: dataField}_\text{Type} \]

The constructor is denoted as

\[ \text{ClassName}(\text{parameter}_\text{Name: parameter}_\text{Type}) \]

The function is denoted as

\[ \text{function}_\text{Name}(\text{parameter}_\text{Name: parameter}_\text{Type}): \text{return}_\text{Type} \]

<PD: UML Class Diagram>
Figure 9.3
Classes and objects can be represented using UML notations.

9.3 Constructors

<Side Remark: overloading constructors>
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 initial data values.

<Side Remark: 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.

<Side Remark: default constructor>
A class may be declared without constructors. In this case, a no-arg constructor with an empty body is implicitly declared in the class. This constructor, called a default constructor, is provided automatically only if no constructors are explicitly declared in the class.

NOTE
Constructors are a special kind of function, with three differences:

<Side Remark: constructor’s name>
• Constructors must have the same name as the class itself.

<Side Remark: no return type>
• Constructors do not have a return type—not even void.

<Side Remark: invoke constructor>
• Constructors are invoked when an object is created. Constructors play the role of initializing objects.

***End of NOTE

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

```cpp
void Circle();
```

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

***End of CAUTION

9.4 Object Names

<side remark: constructing objects>
In C++, you can assign a name when creating an object. A constructor is invoked when an object is created. The syntax to create an object using the no-arg constructor is

\[\text{ClassName \ variableName;}\]

For example, the following declaration creates an object named \texttt{circle1} by invoking the \texttt{Circle} class’s no-arg constructor.

\[
\text{Circle circle1;}
\]

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

\[\text{ClassName \ variableName(arguments);}\]

For example, the following declaration creates an object named \texttt{circle2} by invoking the \texttt{Circle} class’s constructor with a specified radius 5.5.

\[
\text{Circle circle2(5.5);}\]

Newly created objects are allocated in the memory. How can they be accessed? Objects are accessed via object names, which contain references to the objects.

After an object is created, its data can be accessed and its functions invoked using the dot operator (\texttt{.}), also known as the \texttt{object member access operator}:

- \texttt{objectName.dataField} references a data field in the object.
- \texttt{objectName.function(arguments)} invokes a function on the object.

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

\[\text{<Side Remark: instance variable>}\]
\[\text{<Side Remark: instance function>}\]
\[\text{<Side Remark: calling object>}\]

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

324
Listing 9.1 gives a program that demonstrates classes and objects. The program constructs a circle object with radius 1 (line 31) and an object with radius 5 (line 32) and displays the radius and area of each of the two circles (lines 34–37). Change the radius of the second object to 100 (line 40) and display its new radius and area (lines 41–42).

### Listing 9.1 TestCircle.cpp (A Simple Circle Class)

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

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;
        }
};  // Must place a semicolon here

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;
}

<Output>
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
<End Output>

.Side Remark: ending class declaration>
The class is defined in lines 4-27. Don’t forget that the semicolon (;) in line 27 is required.

.Side Remark: public>
The public keyword in line 6 denotes that all data fields, constructors, and functions can be accessed from the objects of the class.

.Side Remark: initialize data field>
The data field radius does not have an initial value. So, it must be initialized in the constructor (lines 13, 19). A primitive data field cannot be declared and initialized at the same time. For example, it would be wrong to replace line 5 by

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

The main function (lines 29-45) creates two objects. The constructor Circle() was used to create circle1 with a radius of 1.0 (line 31), and the constructor Circle(5.0) was used to create circle2 with a radius of 5.0 (line 32).

These two objects (referenced by circle1 and circle2) have different data but share the same functions. Therefore, you can compute their respective areas by using the getArea() function.

To write the getArea function in a procedural programming language like Pascal, you would pass radius as an argument to the function. But in object-oriented programming, radius and getArea are defined for the object. The radius is a data member in a circle object, which is accessible from the getArea function. In procedural programming languages, data and functions are separated, but in an object-oriented programming language, data and functions are grouped together.

The getArea function is an instance function that is always invoked by an instance in which the radius is specified.

NOTE
When you declare 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.

NOTE:

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.

NOTE

In C++, you can also 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,

```cpp
circle2 = circle1;
```

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

NOTE:

Object names are like array names. Once an object name is declared, it references an object. It cannot be reassigned to reference another object. In this sense, an object name is a constant, though the contents of the object may change.

NOTE

Most of the time, you create a named object and later access the members of the object through its name. Occasionally, you may create an object and use it only once. In this case, you don’t have to name the object. 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,

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

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

```cpp
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.

```cpp
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 the object will not be referenced later.

***End of NOTE

CAUTION

<Side Remark: no-arg constructor>

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

TIP

<side remark: debugging in IDE>

If you use an IDE such as C++Builder and Visual C++.NET, please refer to Learning C++ Effectively with C++Builder/Visual C++.NET in the supplements. This supplement shows you how to use a debugger to inspect objects.

NOTE

<Side Remark: class replaces struct>

The C language has the struct type for representing records. For example, you may
define a struct type for representing students as shown in (a).

(a) struct Student {
    int id;
    char firstName[30];
    char mi;
    char lastName[30];
};

(b) class Student {
    public:
    int id;
    char firstName[30];
    char mi;
    char lastName[30];
};

The struct type is replaced by the C++ class. A struct is essentially a class with all public data fields but no functions (although some compilers support functions). The struct type defined in (a) can be replaced by the class defined in (b).

9.5 Separating Interface from Implementation
The preceding example declares the class along with the main function that uses the class in the same file. You cannot reuse the class in other programs. To make the class reusable, you should declare the class in a separate header file.

C++ allows you to separate class interface from implementation. The class interface describes the contract of the class and the class implementation implements the contract. The class interface simply lists all the data fields, constructor prototypes, and the function prototypes. The class implementation implements the constructors and functions. The class interface and implementation are in two separate files. Both files should have the same name, but with different extension names. The class interface file has an extension name .h and the class implementation file has an extension name .cpp.

Listings 9.2 and 9.3 present the Circle class interface and implementation.

Listing 9.2 Circle.h (Circle Class Interface)

***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.***

.Side Remark line 5: data field
.Side Remark line 8: no-arg constructor
.Side Remark line 10: second constructor
.Side Remark line 13: function prototype
.Side Remark line 14: semicolon required
class Circle
{
public:
    // The radius of this circle
    double radius;

    // Construct a circle object
    Circle();
    Circle(double);

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

CAUTION
<Side Remark: don’t omit semicolon>
It is a common mistake to omit the semicolon (;) at the end of the header file.

Listing 9.3 Circle.cpp (Circle Class Implementation)
***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.***
<Side Remark line 1: include class interface>
<Side Remark line 4: implement constructor>
<Side Remark line 10: implement constructor>
<Side Remark line 16: implement function>

#include "Circle.h"

// Construct a 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;
}

<Side Remark: scope resolution operator>
Note that Circle:: preceding each constructors and functions in the Circle class tells the compiler that these are defined in the Circle class. The :: symbol is called the scope resolution operator in C++.

Listing 9.4 gives a client program that uses the Circle class.

```cpp
#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

NOTE

<Side Remark: compiling from command line>

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

```
g++ Circle.cpp TestCircleWithInterface.cpp -o Main
```
If your main program uses the custom defined files, all these files must be present in the project pane in the IDE. Otherwise, you may get linking errors.

Data fields may be initialized in the constructor initialization section using the following syntax:

```
ClassName(parameterList)
  {  // datafield1(value1), datafield2(value2)
    // Additional operations if needed
  }
```

The initialization section initializes `datafield1` with `value1` and `datafield2` with `value2`.

For example,

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

### 9.6 Accessing Object Members via Pointers

Object names cannot be changed once they are declared. However, you can create pointers for object names and assign new object names to pointers whenever necessary. For example,

**PD: Please add line numbers in the following code***

**Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.**

```
<Side Remark line 1: declare object>
Circle circle1;
Circle *pCircle = &circle1;

<Side Remark line 2: assign to pointer>
Line 1 declares a circle object. Line 2 declares a pointer for the circle object. Later you may assign the address of a Circle object to the pointer.
```
To access object members via a pointer, you must dereference the pointer and use the dot (.) operator to object’s members. For example,

```cpp
cout << "The radius is " << (*pCircle).radius << endl;
cout << "The area is " << (*pCircle).getArea() << endl;
(*pCircle).radius = 5.5;
cout << "The radius is " << (*pCircle).radius << endl;
cout << "The area is " << (*pCircle).getArea() << endl;
```

C++ also provides a shorthand member selection operator for accessing object members from a pointer: arrow (->) operator, which is a dash (-) immediately followed by the greater than (> ) symbol. For example,

```cpp
cout << "The radius is " << pCircle -> radius << endl;
cout << "The area is " << pCircle -> getArea() << endl;
pCircle -> radius = 5.5;
cout << "The radius is " << pCircle -> radius << endl;
cout << "The area is " << pCircle -> getArea() << endl;
```

9.7 Creating Dynamic Objects on Heap

When you declare a circle object in a function, it is created in the stack. When the function returns, the object is destroyed. To retain the object, you may create it dynamically on the heap using the `new` operator.

```cpp
ClassName *pObject = new ClassName();
```

creates an object using the no-arg constructor and assigns the object address to the pointer.
ClassName *pObject = new ClassName(arguments);

creates an object using the constructor with arguments and assigns the object address to the pointer.

For example, see the following code:

    // Create an object using the no-arg constructor
    Circle *pCircle1 = new Circle();

    // Create an object using the constructor with arguments
    Circle *pCircle2 = new Circle(5.9);

The objects are destroyed when the program terminated. To explicitly destroy an object, invoke

    delete pObject;

9.8 The C++ string Class
Listing 9.1 declared the Circle class and created objects from the class. You will frequently use the classes in the C++ library to develop programs. This section introduces the string class in the C++ library.

You learned how to store strings using arrays of characters and how to process strings using the C-string functions. C++ also provides the string class. You can use it to create string objects and process strings using its member functions. The string class supports more features than the standard C-string functions.

Some frequently used constructors and functions in the string class are shown in the UML diagram in Figure 9.4.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string()</code></td>
<td>Constructs an empty string.</td>
</tr>
<tr>
<td><code>string(value: string)</code></td>
<td>Constructs a string with the specified string literal value.</td>
</tr>
<tr>
<td><code>string(value: char[])</code></td>
<td>Constructs a string with the specified character array.</td>
</tr>
<tr>
<td><code>string(ch: char, n: int)</code></td>
<td>Constructs a string initialized with the specified character n times.</td>
</tr>
<tr>
<td><code>append(s: string): string</code></td>
<td>Appends string s into this string object.</td>
</tr>
<tr>
<td><code>append(s: string, index: int, n: int): string</code></td>
<td>Appends n number of characters in s starting at the position index to this string.</td>
</tr>
<tr>
<td><code>append(s: char[], n: int): string</code></td>
<td>Appends the first n number of characters in s to this string.</td>
</tr>
<tr>
<td><code>append(n: int, ch: char): string</code></td>
<td>Appends n copies of character ch to this string.</td>
</tr>
<tr>
<td><code>assign(s: string, index: int, n: int): string</code></td>
<td>Assigns n number of characters in s starting at the position index to this string.</td>
</tr>
<tr>
<td><code>assign(s: string, n: int): string</code></td>
<td>Assigns the first n number of characters in s to this string.</td>
</tr>
<tr>
<td><code>assign(s: string, n: int): string</code></td>
<td>Assigns n copies of character ch to this string.</td>
</tr>
<tr>
<td><code>at(index: int): char</code></td>
<td>Assigns array of characters or a string s to this string.</td>
</tr>
<tr>
<td><code>length(): int</code></td>
<td>Returns the character at the position index from this string.</td>
</tr>
<tr>
<td><code>size(): int</code></td>
<td>Returns the number of characters in this string.</td>
</tr>
<tr>
<td><code>capacity(): int</code></td>
<td>Same as length().</td>
</tr>
<tr>
<td><code>clear(): void</code></td>
<td>Returns the size of the storage allocated for this string.</td>
</tr>
<tr>
<td><code>erase(index: int, n: int): string</code></td>
<td>Removes all characters in this string.</td>
</tr>
<tr>
<td><code>empty(): bool</code></td>
<td>Removes n characters from this string starting at position index.</td>
</tr>
<tr>
<td><code>equal(s: string): int</code></td>
<td>Returns true if this string is empty.</td>
</tr>
<tr>
<td><code>compare(s: string): int</code></td>
<td>This two compare functions are like the strcmp function in §7.9.4, “String Functions,” with the same return value.</td>
</tr>
<tr>
<td><code>compare(index: int, n: int, s: string): int</code></td>
<td>Copies n characters into s starting at position index.</td>
</tr>
<tr>
<td><code>copy(s: char[], index: int, n: int): void</code></td>
<td>Returns a character array from this string.</td>
</tr>
<tr>
<td><code>data(): char[]</code></td>
<td>Returns a substring of n characters from this starting at position index.</td>
</tr>
<tr>
<td><code>substr(index: int, n: int): string</code></td>
<td>Returns a substring of this string starting at position index.</td>
</tr>
<tr>
<td><code>swap(s: string): void</code></td>
<td>Swaps this string with s.</td>
</tr>
<tr>
<td><code>find(ch: char): int</code></td>
<td>Returns the position of the first matching character for ch.</td>
</tr>
<tr>
<td><code>find(ch: char, index: int): int</code></td>
<td>Returns the position of the first matching character for ch at or from the position index.</td>
</tr>
<tr>
<td><code>find(s: string): int</code></td>
<td>Returns the position of the first matching substring s.</td>
</tr>
<tr>
<td><code>find(s: string, index: int): int</code></td>
<td>Returns the position of the first matching substring s starting at or from the position index.</td>
</tr>
<tr>
<td><code>replace(index: int, n: int, s: string): string</code></td>
<td>Replaces the n characters starting at position index in this string with the string s.</td>
</tr>
<tr>
<td><code>insert(index: int, s: string): string</code></td>
<td>Inserts the string s into this string at position index.</td>
</tr>
<tr>
<td><code>insert(index: int, n: int, ch: char): string</code></td>
<td>Inserts the character ch n times into this string at position index.</td>
</tr>
</tbody>
</table>

**Figure 9.4**

The **string** class provides the functions for processing a string.

**NOTE**

**<Side Remark: return string>**

Many functions in the string class return a string such as `append`, `assign`, `erase`, `substr`, `replace`, and `insert`. In this case, the resulting new string is returned.

### 9.8.1 Constructing a String
You can create an empty string using string's no-arg constructor like this one:
```cpp
string newString;
```

You can create a string object from a string value or from an array of characters. To create a string from a string literal, use a syntax like this one:
```cpp
string newString(stringLiteral);
```

9.8.2 Appending a String
You can use several overloaded functions to add new contents to a string. For example, see the following code:
```cpp
string s1("Welcome");
s1.append(" to C++"); // appends " to C++" to s1
cout << s1 << endl; // s1 now becomes Welcome to C++

string s2("Welcome");
s2.append(" to C and C++", 0, 5); // appends " to C" to s2
cout << s2 << endl; // s2 now becomes Welcome to C

string s3("Welcome");
s3.append(" to C and C++", 5); // appends " to C" to s3
cout << s3 << endl; // s3 now becomes Welcome to C

string s4("Welcome");
s4.append(4, 'G'); // appends "GGGG" to s4
cout << s4 << endl; // s4 now becomes WelcomeGGGG
```

9.8.3 Assigning a String
You can use several overloaded functions to assign new contents to a string. For example, see the following code:
```cpp
string s1("Welcome");
s1.assign("Dallas"); // assigns "Dallas" to s1
cout << s1 << endl; // s1 now becomes Dallas

string s2("Welcome");
s2.assign("Dallas, Texas", 0, 5); // assigns "Dalla" to s2
cout << s2 << endl; // s2 now becomes Dalla

string s3("Welcome");
s3.assign("Dallas, Texas", 5); // assigns "Dalla" to s3
cout << s3 << endl; // s3 now becomes Dalla

string s4("Welcome");
s4.assign(4, 'G'); // assigns "GGGG" to s4
cout << s4 << endl; // s4 now becomes GGGG
```

### 9.8.4 Functions at, clear, erase, and empty

You can use the `at(index)` function to retrieve a character at a specified index, `clear()` to clear the string, `erase(index, n)` to delete part of the string, and `empty()` to test if a string is empty. For example, see the following code:

```cpp
string s1("Welcome");
cout << s1.at(3) << endl; // s1.at(3) returns c
cout << s1.erase(2, 3) << endl; // s1 is now Weme
s1.clear(); // s1 is now empty
cout << s1.empty() << endl; // s1.empty returns 1 (means true)
```

### 9.8.5 Functions length, size, and capacity

You can use the functions `length()`, `size()`, and `capacity()` to obtain string’s length, size, and capacity. For example, see the following code:

```cpp
***PD: Please add line numbers in the following code***

***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.***

<Side Remark line 1: create string>
<Side Remark line 6: erase two characters>

```cpp
string s1("Welcome");
cout << s1.length() << endl; // length is 7
cout << s1.size() << endl; // size is 7
cout << s1.capacity() << endl; // capacity is 7

s1.erase(1, 2);
cout << s1.length() << endl; // length is now 5
cout << s1.size() << endl; // size is now 5
cout << s1.capacity() << endl; // capacity is still 7
```

**NOTE**

<Side Remark: capacity?>

The capacity is set to 7 when string `s1` is created in line 1. After erasing two characters in line 6, the capacity is still 7, but the length and size become 5.
9.8.6 Comparing Strings
Often, in a program, you need to compare the contents of two strings. You can use the compare function. This function works in the same way as the C-string strcmp function and returns the same a value greater than 0, 0, or less than 0. For example, see the following code:

```cpp
string s1("Welcome");
string s2("Welcomg");
cout << s1.compare(s2) << endl; // returns -2
cout << s2.compare(s1) << endl; // returns 2
cout << s1.compare("Welcome") << endl; // returns 0
```

9.8.7 Obtaining Substrings
You can obtain a single character from a string using the at function. You can also obtain a substring from a string using the substr function. For example, see the following code:

```cpp
string s1("Welcome");
cout << s1.substr(0, 1) << endl; // returns W
cout << s1.substr(3) << endl; // returns come
cout << s1.substr(3, 3) << endl; // returns com
```

9.8.8 Searching in a String
You can use the find function to search for a substring or a character in a string. For example, see the following code:

```cpp
string s1("Welcome to HTML");
cout << s1.find("co") << endl; // returns 3
cout << s1.find("co", 6) << endl; // returns -1
cout << s1.find('o') << endl; // returns 4
cout << s1.find('o', 6) << endl; // returns 9
```

9.8.8 Inserting and Replacing Strings
Here are the examples to use the insert and replace functions:

```cpp
string s1("Welcome to HTML");
s1.insert(11, "C++ and ");
cout << s1 << endl; // s1 becomes Welcome to C++ and HTML

string s2("AA");
s2.insert(1, 4, 'B');
cout << s2 << endl; // s2 becomes to ABBBA

string s3("Welcome to HTML");
s3.replace(11, 4, "C++");
cout << s3 << endl; // returns Welcome to C++
```

9.8.9 String Operators
C++ supports string operators to simplify string operations. Table 9.1 lists the string operators.

**Table 9.1**
String Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>Accesses characters using the array subscript operators.</td>
</tr>
<tr>
<td>=</td>
<td>Copies the contents of one string to the other.</td>
</tr>
<tr>
<td>+</td>
<td>Concatenates two strings into a new string.</td>
</tr>
<tr>
<td>+=</td>
<td>Appends the contents of one string to the other.</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>Inserts a string to a stream</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>Extracts characters from a stream to a string delimited by a whitespace or the null terminator character.</td>
</tr>
<tr>
<td>==, !=, &lt;, &lt;=, &gt;, &gt;=</td>
<td>Six relational operators for comparing strings.</td>
</tr>
</tbody>
</table>

Here are the examples to use these operators:

```cpp
string s1 = "ABC"; // The = operator
string s2 = s1;  // The = operator
for (int i = s2.size() – 1; i >= 0; i--)
    cout << s2[i]; // The [] operator

string s3 = s1 + "DEFG" ; // The + operator
cout << s3 << endl; // s3 becomes ABCDEFG

s1 += "ABC";
cout << s1 << endl; // s1 becomes ABCABC

s1 = "ABC";
s2 = "ABE";
cout << (s1 == s2) << endl; // Displays false
cout << (s1 != s2) << endl; // Displays true
cout << (s1 > s2) << endl; // Displays false
cout << (s1 >= s2) << endl; // Displays false
cout << (s1 < s2) << endl; // Displays true
cout << (s1 <= s2) << endl; // Displays true
```

9.9 Data Field Encapsulation

<Side Remark 1: client>
<Side Remark 2: data field encapsulation>
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.
- 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 non-negative 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. Such programs are often referred to as clients. This is because the clients may have modified the radius directly (e.g., myCircle.radius = -5).

To prevent direct modifications of properties, you should declare the field private, using the private modifier. This is known as data field encapsulation. Making the radius data field private in the Circle class, the class interface can be declared as follows:

```cpp
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 modify a data field. To make a private data field accessible, provide a get function to return the value of the data field. To enable a private data field to be updated, provide a set function to set a new value.

**NOTE**

*<Side Remark 1: accessor>*
*<Side Remark 2: mutator>*

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

A get function has the following signature:

```cpp
returnType getPropertyName()
```

*<Side Remark: bool accessor>*
If the `returnType` is `bool`, the `get` function should be defined as follows by convention:

```cpp
bool isPropertyName()
```

A set function has the following signature:

```cpp
public 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 9.5. The new circle class is declared in Listing 9.5:

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

Figure 9.5
The Circle class encapsulates circle properties and provides get/set and other functions.

Listing 9.5 Circle2.h (A Circle Class with Private Fields)

```cpp
***PD: Please add line numbers in the following code***
<Side Remark line 3: public>
<Side Remark line 7: access function>
<Side Remark line 8: mutator function>
<Side Remark line 10: private>
```

Listing 9.6 implements the class contract specified in Listing 9.5.

Listing 9.6 Circle2.cpp (Implementing Circle2.h)

```cpp
***PD: Please add line numbers in the following code***
<Side Remark line 1: include header file>
<Side Remark line 4: constructor>
<Side Remark line 10: constructor>
```
#include "Circle2.h"

// Construct a 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;
}

// Return the radius of this circle
double Circle::getRadius()
{
    return radius;
}

// Set a new radius
void Circle::setRadius(double newRadius)
{
    radius = (newRadius >= 0) ? newRadius : 0;
}

The \texttt{getRadius()} function (lines 20-25) returns the radius, and the \texttt{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 implementation of these functions, you need not change the client programs. This makes the class easy to maintain.

Listing 9.7 is a client program that uses the Circle class to create a Circle object and modifies the radius using the \texttt{setRadius} function.

Listing 9.7 \texttt{TestCircle2.cpp} (Client for Circle2.h)

***PD: Please add line numbers in the following code***

<Side Remark line 2: include header file>
<Side Remark line 7: construct object>
```cpp
#include <iostream>
#include "Circle2.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;
}
```

**Output**

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

**End Output**

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

Since radius is private, it cannot be modified. This prevents tampering. For example, the user cannot set radius to -100.

**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.

**9.10 The Scope of Variables**

Chapter 5, “Functions,” discussed the scope of global variables and 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.
The data fields are declared as variables and are accessible to all constructors and functions in the class. In this sense, data fields are like global variables. However, data fields and functions can be declared 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
{
 private:
 double radius;
 public:
 Circle();
 Circle(double);
 double getArea();
 double getRadius();
 void setRadius(double);
}
```

(b)

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

(c)

**TIP**

*<side remark: public first>*

Though the class members can be declared in any order, it is better to declare public members first and then the private members, because the clients are not interested in the private members.

Local variables are declared and used inside a function locally. 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.

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 following program in Listing 9.8, `x` is defined as a data field and as a local variable in the function.

Listing 9.8 HideDataField.cpp (Hiding Data Field)

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

class Foo
{
    public:
        Circle();
        Circle(double);
        double getArea();
        double getRadius();
        void setRadius(double);
    private:
        double radius;
};
```

**<Side Remark line 7: data field x>**
**<Side Remark line 8: data field y>**
**<Side Remark line 10: no-arg constructor>**
**<Side Remark line 18: local variable>**
**<Side Remark line 26: create object>**
**<Side Remark line 27: invoke function>**

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

    class Foo
    {
        public:
            Circle();
            Circle(double);
            double getArea();
            double getRadius();
            void setRadius(double);
        private:
            double radius;
    };
```
```cpp
public:
    int x; // data field
    int y; // data field

    Foo()
    {
        x = 10;
        y = 10;
    }

da void p()
{
    int x = 20; // local variable
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
}

int main()
{
    Foo foo;
    foo.p();

    return 0;
}
```

Why the printout is 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 outside the function `p()`, but is accessible inside it.

**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.

### 9.11 The this Pointer

**Side Remark: hidden variable**
Sometimes you need to reference a class’s hidden data field in a function. For example, a property name is often used as the parameter name in a set function for the property. In this case, you need to reference the hidden property name in the function in order to set a new value to it. A hidden data field can be accessed by using the `this` keyword, which

```
is a special built-in pointer that references to the calling object. You can rewrite the Circle class implementation in Listing 9.9 using the \texttt{this} pointer, as shown in Listing 9.11:

Listing 9.9 Circle3.cpp (Implementing Circle2.h)

***PD: Please add line numbers in the following code***
\texttt{<Side Remark line 1: include header file>}
\texttt{<Side Remark line 12: this pointer>}
\texttt{<Side Remark line 30: this pointer>}
\begin{verbatim}
#include "Circle2.h"

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

// Construct a circle object
Circle::Circle(double radius)
{
    this->radius = radius; // or (*this).radius = radius;
}

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

// Return the radius of this circle
double Circle::getRadius()
{
    return radius;
}

// Set a new radius
void Circle::setRadius(double radius)
{
    this->radius = (radius >= 0) ? radius : 0;
}
\end{verbatim}

The parameter name \texttt{radius} in the constructor (line 10) is a local variable. To reference the data field \texttt{radius} in the object, you have to use \texttt{this->radius} (line 12). The parameter name \texttt{radius} in the \texttt{setRadius} function (line 28) is a local variable. To reference the data field \texttt{radius} in the object, you have to use \texttt{this->radius} (line 30).

\subsection*{9.12 Passing Objects to Functions}
So far, you have learned how to pass arguments of primitive types and array types to functions. You can also pass objects to functions. You can pass objects by value or by
reference. Listings 9.10 gives an example that passes an object by value.

Listing 9.10 PassObjectByValue.cpp (Passing Object By Value)

```cpp
#include <iostream>
#include "Circle2.h"
using namespace std;

void printCircle(Circle c)
{
    cout << "The area of the circle of " << c.getRadius() << " is " << c.getArea() << "\n;"
}

int main()
{
    Circle myCircle(5.0);
    printCircle(myCircle);
    return 0;
}
```

The area of the circle of 5 is 78.5397

The Circle class defined Circle2.h from Listing 9.5 is included in line 2. The parameter for the printCircle function is declared as Circle (line 5). The main function creates a Circle object myCircle (line 13) and passes it to the printCircle function by value (line 14). Passing an object argument by value is to copy the object to the function parameter. So the object c in the printCircle function has the same content as the object myCircle in the main function, as shown in Figure 9.6(a).

![Figure 9.6](image-url)
You can pass an object to a function in three ways: (a) pass by value, (b) pass by reference, (c) pass by reference via pointer.

Listings 9.11 gives an example that passes an object by reference.

Listing 9.11 PassObjectByReference.cpp (Passing Object By Reference)

```cpp
***PD: Please add line numbers in the following code***
<Side Remark line 2: include header file>
<Side Remark line 5: object parameter>
<Side Remark line 8: access circle>
<Side Remark line 13: create circle>
<Side Remark line 14: pass object>

#include <iostream>
#include "Circle2.h"
using namespace std;

void printCircle(const Circle &c)
{
    cout << "The area of the circle of "
    << c.getRadius() << " is " << c.getArea() << endl;
}

int main()
{
    Circle myCircle(5.0);
    printCircle(&myCircle);
    return 0;
}
```

The area of the circle of 5 is 78.5397

A reference parameter of the Circle type is declared in the printCircle function (line 5). The main function creates a Circle object myCircle (line 13) and passes the reference of the object to the printCircle function (line 14). So the object c in the printCircle function is essentially an alias of the object myCircle in the main function, as shown in Figure 9.6(b).

Listings 9.12 gives an example that passes an object by reference via a pointer.

Listing 9.12 PassObjectToPointer.cpp (Passing To Pointer)

```cpp
***PD: Please add line numbers in the following code***
<Side Remark line 2: include header file>
<Side Remark line 5: object parameter>
<Side Remark line 8: access circle>
<Side Remark line 13: create circle>
```
#include <iostream>
#include "Circle2.h"
using namespace std;

void printCircle(const Circle *c)
{
    cout << "The area of the circle of " << c->getRadius() << " is " << c->getArea() << endl;
}

int main()
{
    Circle myCircle(5.0);
    printCircle(&myCircle);
    return 0;
}

The area of the circle of 5 is 78.5397

A pointer parameter of the Circle type is declared in the printCircle function (line 5). The main function creates a Circle object myCircle (line 13) and passes the address of the object to the printCircle function (line 14). So the object c in the printCircle function points to the object myCircle in the main function, as shown in Figure 9.6(c).

TIP

Though you can pass an object to a function by value, by reference, or by reference via a pointer parameter, passing object by reference is preferred for two reasons: (1) It takes time and additional memory space to pass object by value; (2) Using object reference parameter is simpler than using a pointer parameter.

9.13 Array of Objects

In Chapter 6, “Arrays,” arrays of primitive type elements were created. You can also create arrays of objects. For example, the following statement declares and creates an array of ten Circle objects:

Circle circleArray[10]; // Declare array of ten Circle objects

The name of the array is circleArray and the no-arg constructor is called to initialize each element in the array. So, circleArray[0].getRadius() returns 1, because the no-arg constructor assigns 1 to radius.
You can also use the array initializer to declare and initialize an array using a constructor with arguments. For example,

```
Circle circleArray[3] = {Circle(3), Circle(4), Circle(5)};
```

Listing 9.13 gives an example that demonstrates how to use an array of objects. The program summarizes the areas of an array of circles. The program creates `circleArray`, an array composed of ten Circle objects; it then sets circle radii with radius 1, 2, 3, 4, ..., and 10, and displays the total area of the circles in the array.

Listing 9.13 TotalArea.cpp (Array of Objects)

```cpp
***PD: Please add line numbers in the following code***
<Side Remark line 2: include header file>
<Side Remark line 6: array of objects>
<Side Remark line 13: get area>
<Side Remark line 19: array of objects>
<Side Remark line 40: create array>
<Side Remark line 44: new radius>
<Side Remark line 47: pass array>

#include <iostream>
#include "Circle2.h"
using namespace std;

// Add circle areas
double sum(Circle circleArray[], int size)

    // Initialize sum
    double sum = 0;

    // Add areas to sum
    for (int i = 0; i < size; i++)
        sum += circleArray[i].getArea();

    return sum;

// Print an array of circles and their total area
void printCircleArray(Circle circleArray[], int size)

    // Print an array of circles and their total area
    for (int i = 0; i < size; i++)
        cout << circleArray[i].getRadius() << "\t" << circleArray[i].getArea() << endl;

    cout << "-----------------------------------------" << endl;

    // Compute and display the result
    cout << "The total areas of circles is \t" << sum(circleArray, size) << endl;
```
int main()
{
    const int SIZE = 10;

    // Create a Circle object with radius 1
    Circle circleArray[SIZE];

    for (int i = 0; i < SIZE; i++)
    {
        circleArray[i].setRadius(i + 1);
    }

    printCircleArray(circleArray, SIZE);

    return 0;
}

<Output>
Radius     Area
1    3.14159
2    12.5664
3    28.2743
4    50.2654
5    78.5397
6   113.097
7   153.938
8   201.062
9   254.469
10  314.159
---------
The total areas of circles is 1209.51
<End Output>

The program creates an array of ten Circle objects (line 40). Two Circle classes were introduced in this chapter. This example uses the Circle class introduced in §9.9, "Data Field Encapsulation."

Each object element in the array is created using the Circle’s no-arg constructor. A new radius for each circle is set in lines 42-45. circleArray[i] refers to a Circle object in the array. circleArray[i].setRadius(i + 1) sets a new radius in the Circle object (line 44). The array is passed to the printCircleArray function, which displays the radii of the total area of the circles.

The sum of the areas of the circle is computed using the sum function (line 32), which takes the array of Circle objects as the argument and returns a double value for the total area.

9.14 Class Abstraction and Encapsulation

<Side Remark 1: class abstraction>
<Side Remark 2: class encapsulation>
In Chapter 5, "Functions," you learned about function abstraction and used it in 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 the class 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.

![Figure 9.7](image)

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

Class abstraction and encapsulation are two sides of the same coin. There are many real-life examples that illustrate the concept of class abstraction. Consider building a computer system, for instance. 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, and functions like start, stop, and so on. A specific fan is an instance of this class with specific property values.

Consider getting a loan, for another example. 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.
9.15 Case Study: The Loan Class

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.

<PD: UML Class Diagram>

<table>
<thead>
<tr>
<th>Loan</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-annualInterestRate: double</td>
</tr>
<tr>
<td></td>
<td>-numberOfYears: int</td>
</tr>
<tr>
<td></td>
<td>-loanAmount: double</td>
</tr>
<tr>
<td></td>
<td>+Loan()</td>
</tr>
<tr>
<td></td>
<td>+Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double)</td>
</tr>
<tr>
<td></td>
<td>+getAnnualInterestRate(): double</td>
</tr>
<tr>
<td></td>
<td>+getNumberOfYears(): int</td>
</tr>
<tr>
<td></td>
<td>+getLoanAmount(): double</td>
</tr>
<tr>
<td></td>
<td>+setAnnualInterestRate(annualInterestRate: double): void</td>
</tr>
<tr>
<td></td>
<td>+setNumberOfYears(numberOfYears: int): void</td>
</tr>
<tr>
<td></td>
<td>+setLoanAmount(loanAmount: double): void</td>
</tr>
<tr>
<td></td>
<td>+getMonthlyPayment(): double</td>
</tr>
<tr>
<td></td>
<td>+getTotalPayment(): double</td>
</tr>
</tbody>
</table>

-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(annualInterestRate: double, numberOfYears: int, loanAmount: 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(annualInterestRate: double): void
  Sets a new annual interest rate to this loan.
+setNumberOfYears(numberOfYears: int): void
  Sets a new number of years to this loan.
+setLoanAmount(loanAmount: 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 writer. 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 9.14. Let us begin by writing a test program that uses the Loan class in Listing 9.15.

Listing 9.14 Loan.h (Loan Class Header File)
***PD: Please add line numbers in the following code***

<Side Remark line 3: public functions>
<Side Remark line 16: private fields>

class Loan
{
    public:
        Loan();
        Loan(double annualInterestRate, int numberOfYears, double loanAmount);
        double getAnnualInterestRate();
        int getNumberOfYears();
        double getLoanAmount();
}
```cpp
#include <iostream>
#include <iomanip>
#include "Loan.h"
using namespace std;

int main()
{
  // Enter annual interest rate
  cout << "Enter yearly interest rate, for example 8.25: ";
  double annualInterestRate;
  cin >> annualInterestRate;

  // Enter number of years
  cout << "Enter number of years as an integer, for example 5: ";
  int numberOfYears;
  cin >> numberOfYears;

  // Enter loan amount
  cout << "Enter loan amount, for example 120000.95: ";
  double loanAmount;
  cin >> loanAmount;

  // Create Loan object
  Loan loan(annualInterestRate, numberOfYears, loanAmount);

  // Display results
  cout << fixed << setprecision(2);
  cout << "The monthly payment is " << loan.getMonthlyPayment() << endl;
  cout << "The total payment is " << loan.getTotalPayment() << endl;

  return 0;
}
```

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 28) and total payment (line 29) using the instance functions in the Loan class.

The Loan class can be implemented in Listing 9.16.

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

Loan::Loan()
{
  _annualInterestRate = 9.5;
  _numberOfYears = 30;
  _loanAmount = 100000;
}

Loan::Loan(double annualInterestRate, int numberOfYears, double loanAmount)
{
  this->annualInterestRate = annualInterestRate;
  this->numberOfYears = numberOfYears;
  this->loanAmount = loanAmount;
}

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

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

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

void Loan::setAnnualInterestRate(double annualInterestRate)
{
}
```

***PD: Please add line numbers in the following code***

<Side Remark line 5: no-arg constructor>
<Side Remark line 12: constructor>
<Side Remark line 15: this pointer>
<Side Remark line 20: accessor function>
<Side Remark line 25: accessor function>
<Side Remark line 30: accessor function>
<Side Remark line 40: accessor function>
<Side Remark line 45: accessor function>
<Side Remark line 50: compute monthly payment>
<Side Remark line 57: compute total payment>
```cpp
    this->annualInterestRate = annualInterestRate;
}

void Loan::setNumberOfYears(int numberOfYears)
{
    this->numberOfYears = numberOfYears;
}

void Loan::setLoanAmount(double loanAmount)
{
    this->loanAmount = loanAmount;
}

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

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 makes it possible 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 and then turn your attention to its implementation.

End of the TIP

Key Terms

***PD: Please place terms in two columns same as in intro5e.

- accessor function (getter) 225
- class 214
- class abstraction 238
- class encapsulation 238
- class’s contract 238
- constructor 215
- data field encapsulation 225
- default constructor 215
- dot operator (.) 217
- arrow operator (->) 217
- instance 216
- instance function 217
- instance variable 217
- instantiation 216
- mutator function (setter) 225
- no-arg constructor 215
- object-oriented programming (OOP) 214
- Unified Modeling Language (UML) 216
- private 223
- public 223
- scope resolution operator
- this keyword 234

Chapter Summary

- A class is a template for objects. It defines the generic properties of objects, and provides constructors for creating objects and functions for manipulating them.
- A class is also a data type. You can use it to declare object names. An object is an instance of a class. You use the dot (.) operator to access members of that object through its name.
- You can separate class interface from class implementation by defining class interface in a header file and class implementation in a separate file.
- Visibility keywords specify how the class, function, and data are accessed. A public function or data is
accessible to all clients. A private function or data is only accessible inside the class.

• You can provide a get function or a set function to enable clients to see or modify the data. Colloquially, a get function is referred to as a getter (or accessor), and a set function is referred to as a setter (or mutator).

• A get function has the signature returnType getPropertyName(). If the returnType is bool, the get function should be defined as bool isPropertyName(). A set function has the signature void setPropertyName(dataType propertyValue).

• The scope of data fields is the entire class, regardless of where the data fields are declared. The class members can be declared anywhere in the class.

• The keyword this can be used to refer to the calling object.

• You can pass an object to a function by value, by reference, or by reference via a pointer. For performance and simplicity, passing by reference is preferred.

Review Questions
Sections 9.2-9.4
9.1 Describe the relationship between an object and its defining class. How do you declare 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 declare an object using a no-arg constructor? How do you declare 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 What is wrong in the following code? (Use the Circle class defined in Listing 9.1, TestCircle.cpp)

***PD: Please add line numbers in the following code***

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

(b) 
```cpp
int main()
{
    Circle c1(5);
    Circle c1(6);
}
```
9.6
What is wrong in the following code?

***PD: Please add line numbers in the following code***

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

private:
    double radius = 1;
};

Sections 9.5 Separating Interface from Implementation

9.7
How do you separate class interface from implementation?

9.8
What is output of the following code? (Use the Circle class defined in Listing 9.5, Circle2.h)

***PD: Please add line numbers in the following code***

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

(a)            (b)

9.9
Suppose class Time has three data fields: hour, minute, and second. Rewrite the following constructor implementation using the constructor initialization section?

```
Time::Time()
{
    hour = 1;
    minute = 1;
    second = 1;
}
```

Sections 9.6 Accessing Object Members via Pointers

9.10
What is wrong in the following code?

***PD: Please add line numbers in the following code***

```cpp
int main()
{
    Circle c1;
    Circle *p = c1;
}
```

(a)            (b)

9.11
How do you create an object dynamically and how do you delete an object?

Sections 9.8 The C++ string Class

9.12
Suppose that s1 and s2 are two strings, given as follows:

```cpp
string s1("I have a dream");
string s2("Computer Programming");
```

Assume that each expression is independent. What are the results of the following expressions?

1. `s1.append(s2)`
2. `s1.append(s2, 9, 7)`
3. `s1.append("NEW", 3)`
4. `s1.append(3, 'N')`
5. `s1.assign(3, 'N')`
6. `s1.assign(s2, 9, 7)`
7. `s1.assign("NEWNEW", 3)`
8. `s1.assign(3, 'N')`
9. `s1.at(0)`
10. `s1.length()`
11. `s1.size()`
12. `s1.capacity()`
13. `s1.erase(1, 2)`
14. `s1.compare(s3)`
15. `s1.compare(0, 10, s3)`
16. `s1.data()`
17. `s1.substr(4, 8)`
18. `s1.substr(4)`
19. `s1.find('A')`
20. `s1.find('a', 9)`
21. `s1.replace(2, 4, "NEW")`
22. `s1.insert(4, "NEW")`
23. `s1.insert(6, 8, 'N')`
24. `s1.empty()`

9.13
Suppose that s1 and s2 are given as follows:

```cpp
string s1("I have a dream");
string s2("Computer Programming");
char s3[] = "ABCDEFGHIJKLMN";
```

Assume that each expression is independent. What are the results of s1, s2, and s3 after each of the following statements?

1. `s1.clear()`
2. `s1.copy(s3, 5, 2)`
3. `s1.swap(s2)`

9.14
Suppose that s1 and s2 are given as follows:

```cpp
string s1("I have a dream");
string s2("Computer Programming");
```

Assume that each expression is independent. What are the results of the following expressions?

1. `s1[0]`
2. `s1 = s2`
3. `s1 = "C++ " + s2`
4. `s2 += "C++ "`
5. `s1 > s2`
6. `s1 >= s2`
7. `s1 < s2`
8. `s1 <= s2`
9. `s1 == s2`
10. `s1 != s2`

Section 9.9 Data Field Encapsulation

9.15
What is an accessor function? What is a mutator function? What are the naming conventions for accessor functions and mutator functions?

9.16
What are the benefits of data field encapsulation?
Section 9.10 The Scope of Variables

9.17
Can data fields and functions be declared in any order in a class?

Section 9.11 The this Pointer

9.18
What is wrong in the following code? How can it be fixed?

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

Section 9.12 Passing Objects to Functions

9.19
Describe the three ways to pass an object to a function.

Section 9.13 Array of Objects

9.20
How do you declare an array of 10 string objects?

9.21
What is the output in the following code?

```cpp
***PD: Please add line numbers in the following code***
int main()
{
    string cities[] = {"Atlanta", "Dallas", "Savannah"};
    cout << cities[0] << endl;
    cout << cities[1] << endl;
}
```

Programming Exercises

Pedagogical NOTE

<Side Remark: three objectives>
The exercises in Part II achieve three objectives:

1. Design and draw UML for classes;
2. Implement classes from the UML;
3. Use classes to develop applications.

Solutions for the UML diagrams for the even-numbered exercises can be downloaded from the Student Website and all others can be downloaded from the Instructor Website.

Sections 9.2-9.11

9.1
(The Rectangle class) Design a class named Rectangle to represent a rectangle. The class contains:

- Two double data fields named width and height that specify the width and height of the rectangle. The default values are 1 for both width and height.
- A no-arg constructor that creates a default rectangle.
- A constructor that creates a rectangle with the specified width and height.
- The accessor and mutator functions for all three data fields.
- A function named getArea() that returns the area of this rectangle.
- A function named getPerimeter() that returns the perimeter.

Draw the UML diagram for the class. Implement the class.

Write a test program that creates two Rectangle objects. Assign width 4 and height 40 to the first object and width 3.5 and height 35.9 to the second object. Display the properties of both objects and find their areas and perimeters.

9.2
(The Fan class) Design a class named Fan to represent a fan. The class contains:

- Three constants named SLOW, MEDIUM, and FAST with values 1, 2, and 3 to denote the fan speed.
- An int data field named speed that specifies the speed of the fan (default SLOW).
- A bool data field named on that specifies whether the fan is on (default false).
- A double data field named radius that specifies the radius of the fan (default 5).
- A string data field named color that specifies the color of the fan (default blue).
- A no-arg constructor that creates a default fan.
- The accessor and mutator functions for all four data fields.
- A function named toString() that returns a string description for the fan. If the fan is on, the function returns the fan speed, color, and radius in one combined string. If the fan is not on, the function returns fan color and radius along with the string “fan is off” in one combined string.

Draw the UML diagram for the class. Implement the class.

Write a test program that creates two Fan objects. Assign maximum speed, radius 10, color yellow, and turn it on to the first object. Assign medium
speed, radius 5, color blue, and turn it off to
the second object. Display the objects by invoking
their toString function.

9.3
(The Account class) Design a class named Account that
contains:
- An int data field named id for the account (default 0).
- A double data field named balance for the account
  (default 0).
- A double data field named annualInterestRate that
  stores the current interest rate (default 0).
- A no-arg constructor that creates a default account.
- The accessor and mutator functions for id, balance, and
  annualInterestRate.
- The accessor function for dateCreated.
- A function named getMonthlyInterestRate() that returns
  the monthly interest rate.
- A function named withdraw that withdraws a specified
  amount from the account.
- A function named deposit that deposits a specified
  amount to the account.

Draw the UML diagram for the class. Implement the class.
Write a test program that creates an Account
object with an account ID of 1122, a balance of
20000, and an annual interest rate of 4.5%. Use
the withdraw function to withdraw $2500, use the
deposit function to deposit $3000, and print the
balance, the monthly interest, and the date when
this account was created.

9.4
(The Stock class) Design a class named Stock that contains:
- A string data field named symbol for the stock’s
  symbol.
- A string data field named name for the stock’s name.
- A double data field named previousClosingPrice that
  stores the stock price for the previous day.
- A double data field named currentPrice that stores the
  stock price for the current time.
- A constructor that creates a stock with specified
  symbol and name.
- The accessor functions for all data fields.
- The mutator functions for previousClosingPrice and
  currentPrice.
- A function named changePercent() that returns the
  percentage changed from previousClosingPrice to
currentPrice.
Draw the UML diagram for the class. Implement the class. Write a test program that creates a Stock object with the stock symbol SUNW, the name Sun Microsystems Inc, and the previous closing price of 100. Set a new current price to 90 and display the price-change percentage.

9.5* (The Time class) Design a class named Time. The class contains:

- Data fields hour, minute, and second that represents a time.
- A no-arg constructor that creates a Time object for the current time. (The data fields value will represent the current time)
- A constructor that constructs a Time object with a specified elapse time since the middle night, Jan 1, 1970 in milliseconds. (The data fields value will represent this time)
- Three get functions for the data fields hour, minute, and second, respectively.

Draw the UML diagram for the class. Implement the class. Write a test program that creates two Time objects (using new Time() and new Time(555550000)) and display their hour, minute, and second.

Hint: The current time can be obtained using time(0), as shown in Listing 2.11, ShowCurrentTime.cpp. The other constructor sets the hour, minute, and second for the specified elapse time. For example, if the elapse time is 555550 seconds, the hour is 10, the minute is 19, and the second is 10.

9.6 (The MyPoint class) Design a class named MyPoint to represent a point with x and y-coordinates. The class contains:

- Two data fields x and y that represent the coordinates.
- A no-arg constructor that creates a point (0, 0)
- A constructor that constructs a point with specified coordinates.
- Two get functions for data fields x and y, respectively.
- A function named distance that returns the distance from this point to another point of the MyPoint type.
- A function named distance that returns the distance from this point to another point with specified x and y-coordinates.
Draw the UML diagram for the class. Implement the class. Write a test program that creates two points (0, 0) and (10, 30.5) and displays the distance between the two points.

9.7***
(The Tax class) Design a class named Tax to contain the following instance data fields:

- **int filingStatus:** One of the four tax filing statuses: 0 - single filer, 1 - married filing jointly, 2 - married filing separately, and 3 - head of household. Use the public static constants SINGLE_FILER (0), MARRIED_JOINTLY (1), MARRIED_SEPARATELY (2), HEAD_OF_HOUSEHOLD (3) to represent the status.

- **int[][] brackets:** Stores the tax brackets for each filing status (see Listing 5.4, ComputeTaxWithFunction.cpp).

- **double[] rates:** Stores tax rates for each bracket (see Listing 5.9).

- **double taxableIncome:** Stores the taxable income.

Provide the get and set functions for each data field and the getTax() function that returns the tax. Also provide a no-arg constructor and the constructor Tax(filingStatus, brackets, rates, taxableIncome).

Draw the UML diagram for the class. Implement the class. Write a test program that uses the Tax class to print the 2001 and 2002 tax tables for taxable income from $50,000 to $60,000 with intervals of $1000 for all four statuses. The tax rates for the year 2002 were given in Table 3.9. The tax rates for 2001 are shown in Table 9.1.

***Same as Table 9.1 in intro5e p303

Table 9.1
2001 United States Federal Personal Tax Rates

<table>
<thead>
<tr>
<th>Tax rate</th>
<th>Single filers</th>
<th>Married filing jointly or qualifying widow(er)</th>
<th>Married filing separately</th>
<th>Head of household</th>
</tr>
</thead>
<tbody>
<tr>
<td>15%</td>
<td>Up to $27,050</td>
<td>Up to $45,200</td>
<td>Up to $32,600</td>
<td>Up to $36,250</td>
</tr>
<tr>
<td>25.5%</td>
<td>$27,051 - $65,550</td>
<td>$45,201 - $109,250</td>
<td>$22,601 - $34,625</td>
<td>$36,251 - $99,650</td>
</tr>
<tr>
<td>30.5%</td>
<td>$65,551 - $136,750</td>
<td>$109,251 - $166,500</td>
<td>$54,626 - $83,250</td>
<td>$93,651 - $151,650</td>
</tr>
<tr>
<td>35.5%</td>
<td>$136,751 - $297,350</td>
<td>$166,501 - $297,350</td>
<td>$83,251 - $149,675</td>
<td>$151,651 - $297,350</td>
</tr>
<tr>
<td>39.1%</td>
<td>$297,351 or more</td>
<td>$297,351 or more</td>
<td>$146,676 or more</td>
<td>$297,351 or more</td>
</tr>
</tbody>
</table>