# Introduction to Classes, Objects, Member Functions and Strings

3

## Objectives

In this chapter you'll:

- Begin programming with the object-oriented concepts introduced in Section 1.8.
- Define a class and use it to create an object.
- Implement a class's behaviors as member functions.
- Implement a class's attributes as data members.
- Call an object's member functions to make them perform their tasks.
- Access and manipulate `private` data members through their corresponding `public` *get* and *set* functions to enforce encapsulation of the data.
- Learn what local variables of a member function are and how they differ from data members of a class.
- Use a constructor to initialize an object's data.
- Validate the data passed to a constructor or member function.
- Become familiar with UML class diagrams.

Outline

## 3.1 Introduction[1]

Section 1.8 presented a friendly introduction to object orientation, discussing classes, objects, data members (attributes) and member functions (behaviors).[2] In this chapter's examples, we make those concepts real by building a simple bank-account class. The class maintains as *data members* the attributes `name` and `balance`, and provides *member functions* for behaviors including

- querying the balance (`getBalance`),
- making a deposit that increases the balance (`deposit`) and
- making a withdrawal that decreases the balance (`withdraw`).

We'll build the `getBalance` and `deposit` member functions into the chapter's examples. You'll add the `withdraw` member function in Exercise 3.9.

As you'll see, each *class* you create becomes a *new type* you can use to create objects, so C++ is an extensible programming language. If you become part of a development team in industry, you might work on applications that contain hundreds, or even thousands, of custom classes.

1. This chapter depends on the terminology and concepts introduced in Section 1.8, Introduction to Object Technology.
2. Unlike classes, fundamental types (like `int`) do not have member functions.

## 3.2 Test-Driving an Account Object

Classes *cannot* execute by themselves. A `Person` object can drive a `Car` object by telling it what to do (go faster, go slower, turn left, turn right, etc.)—without knowing how the car's internal mechanisms work. Similarly, the `main` function can "drive" an `Account` object by calling its member functions—without knowing how the class is implemented. In this sense, `main` is referred to as a **driver program**. We show the `main` program and its output first, so you can see an `Account` object in action. To help you prepare for the larger programs you'll encounter later in this book and in industry, we define `main` in its own file (file `AccountTest.cpp`, Fig. 3.1). We define class `Account` in its own file as well (file `Account.h`, Fig. 3.2).

```
 1  // Fig. 3.1: AccountTest.cpp
 2  // Creating and manipulating an Account object.
 3  #include <iostream>
 4  #include <string>
 5  #include "Account.h"
 6
 7  using namespace std;
 8
 9  int main() {
10     Account myAccount; // create Account object myAccount
11
12     // show that the initial value of myAccount's name is the empty string
13     cout << "Initial account name is: " << myAccount.getName();
14
15     // prompt for and read name
16     cout << "\nPlease enter the account name: ";
17     string theName;
18     getline(cin, theName); // read a line of text
19     myAccount.setName(theName); // put theName in myAccount
20
21     // display the name stored in object myAccount
22     cout << "Name in object myAccount is: "
23        << myAccount.getName() << endl;
24  }
```

```
Initial account name is:
Please enter the account name: Jane Green
Name in object myAccount is: Jane Green
```

**Fig. 3.1** | Creating and manipulating an `Account` object.

### 3.2.1 Instantiating an Object

Typically, you cannot call a member function of a class until you *create an object* of that class.[3] Line 10

```
   Account myAccount; // create Account object myAccount
```

creates an object of class `Account` called `myAccount`. The variable's type is `Account`—the class we define in Fig. 3.2.

3. You'll see in Section 9.15 that `static` member functions are an exception.

## 3.2.2 Headers and Source-Code Files

When we declare variables of type `int`, as we did in Chapter 2, the compiler knows what `int` is—it's a *fundamental type* that's "built into" C++. In line 10, however, the compiler does *not* know in advance what type `Account` is—it's a **user-defined type**.

When packaged properly, new classes can be *reused* by other programmers. It's customary to place a reusable class definition in a file known as a **header** with a `.h` filename extension.[4] You include (via `#include`) that header wherever you need to use the class. For example, you can *reuse* the C++ Standard Library's classes in any program by including the appropriate headers.

Class `Account` is defined in the header `Account.h` (Fig. 3.2). We tell the compiler what an `Account` is by including its header, as in line 5 (Fig. 3.1):

```
#include "Account.h"
```

If we omit this, the compiler issues error messages wherever we use class `Account` and any of its capabilities. In an `#include` directive, a header that *you* define in *your program* is placed in double quotes (`""`), rather than the angle brackets (`<>`) used for C++ Standard Library headers like `<iostream>`. The double quotes in this example tell the compiler that header is in the same folder as Fig. 3.1, rather than with the C++ Standard Library headers.

Files ending with the `.cpp` filename extension are **source-code files**. These define a program's `main` function, other functions and more, as you'll see in later chapters. You include headers into source-code files (as in Fig. 3.1), though you also may include them in other headers.

## 3.2.3 Calling Class Account's getName Member Function

The `Account` class's `getName` member function returns the account name stored in a particular `Account` object. Line 13

```
cout << "Initial name is: " << myAccount.getName();
```

displays `myAccount`'s *initial* name by calling the object's `getName` member function with the expression `myAccount.getName()`. To call this member function for a specific object, you specify the object's name (`myAccount`), followed by the **dot operator** (`.`), then the member function name (`getName`) and a set of parentheses. The *empty* parentheses indicate that `getName` does not require any additional information to perform its task. Soon, you'll see that the `setName` function requires additional information to perform its task.

From `main`'s view, when the `getName` member function is called:

1. The program transfers execution from the call (line 13 in `main`) to member function `getName`. Because `getName` was called via the `myAccount` object, `getName` "knows" which object's data to manipulate.
2. Next, member function `getName` performs its task—that is, it *returns* (i.e., gives back) `myAccount`'s name to line 13 where the function was called. The main function does not know the details of how `getName` performs its task.
3. The `cout` object displays the name returned by member function `getName`, then the program continues executing at line 16 in `main`.

4. C++ Standard Library headers, like `<iostream>` do not use the `.h` filename extension.

In this case, line 13 does not display a name, because we have not yet stored a name in the `myAccount` object.

## 3.2.4 Inputting a `string` with `getline`

Line 17

```
    string theName;
```

creates a **`string`** variable called `theName` that's used to store the account name entered by the user. `string` variables can hold character string values such as `"Jane Green"`. A `string` is actually an *object* of the C++ Standard Library class `string`, which is defined in the **header `<string>`**.[5] The class name `string`, like the name `cout`, belongs to namespace `std`. To enable line 17 to compile, line 4 includes the `<string>` header. The `using` directive in line 7 allows us to write `string` in line 17 rather than `std::string`.

### *`getline` Function Receiving a Line of Text from the User*

Sometimes functions are *not* members of a class. Such functions are called **global functions**. Line 18

```
    getline(cin, theName); // read a line of text
```

reads the name from the user and places it in the variable `theName`, using the C++ Standard Library global function **`getline`** to perform the input. Like class `string`, function `getline` requires the `<string>` header and belongs to namespace `std`.

Consider why we cannot simply write

```
    cin >> theName;
```

to obtain the account name. In our sample program execution, we entered the name "`Jane Green`," which contains multiple words *separated by a space*. (Recall that we highlight user inputs in bold in our sample program executions.) When reading a `string`, `cin` stops at the first *white-space character* (such as a space, tab or newline). Thus, the preceding statement would read only `"Jane"`. The information after `"Jane"` is *not lost*—it can be read by subsequent input statements later in the program.

In this example, we'd like the user to type the complete name (including the space) and press *Enter* to submit it to the program. Then, we'd like to store the *entire* name in the `string` variable `theName`. When you press *Enter* (or *Return*) after typing data, the system inserts a newline in the input stream. Function `getline` reads from the standard input stream object `cin` the characters the user enters, up to, but *not* including, the newline, which is *discarded*; `getline` places the characters in the `string` variable `theName`.

## 3.2.5 Calling Class `Account`'s `setName` Member Function

The `Account` class's `setName` member function stores an account name in a particular `Account` object. Line 19

```
    myAccount.setName(theName); // put theName in myAccount
```

5. You'll learn additional `string` capabilities in subsequent chapters. Chapter 21 discusses class `string` in detail, presenting many of its member functions.

calls `myAccounts`'s `setName` member function. A member-function call can supply **arguments** that help the function perform its task. You place the arguments in the function call's parentheses. Here, `theName`'s value (input by line 18) is the *argument* that's passed to `setName`, which stores `theName`'s value in the object `myAccount`.

From `main`'s view, when `setName` is called:

1. The program transfers execution from line 19 in `main` to `setName` member function's definition. The call passes to the function the *argument value* in the call's parentheses—that is, `theName` object's value. Because `setName` was called via the `myAccount` object, `setName` "knows" the exact object to manipulate.
2. Next, member function `setName` stores the argument's value in the `myAccount` object.
3. When `setName` completes execution, program execution returns to where `setName` was called (line 19), then continues at line 22.

### *Displaying the Name That Was Entered by the User*

To demonstrate that `myAccount` now contains the name the user entered, lines 22–23

```
    cout << "Name in object myAccount is: "
       << myAccount.getName() << endl;
```

call member function `getName` again. As you can see in the last line of the program's output, the name entered by the user in line 18 is displayed. When the preceding statement completes execution, the end of `main` is reached, so the program terminates.

## 3.3 Account Class with a Data Member and *Set* and *Get* Member Functions

Now that we've seen class `Account` in action (Fig. 3.1), we present class `Account`'s details. Then, we present a UML diagram that summarizes class `Account`'s *attributes* and *operations* in a concise graphical representation.

### 3.3.1 Account Class Definition

Class `Account` (Fig. 3.2) contains a `name` *data member* that stores the account holder's name. A class's data members maintain data for each object of the class. Later in the chapter, we'll add a `balance` data member to keep track of the money in each `Account`. Class `Account` also contains member function `setName` that a program can call to store a name in an `Account` object, and member function `getName` that a program can call to obtain a name from an `Account` object.

```
1  // Fig. 3.2: Account.h
2  // Account class that contains a name data member
3  // and member functions to set and get its value.
4  #include <string> // enable program to use C++ string data type
5
```

**Fig. 3.2** | `Account` class that contains a `name` data member and member functions to *set* and *get* its value. (Part 1 of 2.)

```
 6  class Account {
 7  public:
 8     // member function that sets the account name in the object
 9     void setName(std::string accountName) {
10        name = accountName; // store the account name
11     }
12
13     // member function that retrieves the account name from the object
14     std::string getName() const {
15        return name; // return name's value to this function's caller
16     }
17  private:
18     std::string name; // data member containing account holder's name
19  }; // end class Account
```

**Fig. 3.2** | `Account` class that contains a `name` data member and member functions to *set* and *get* its value. (Part 2 of 2.)

### 3.3.2 Keyword `class` and the Class Body

The *class definition* begins in line 6:

```
      class Account {
```

Every class definition contains the keyword `class` followed immediately by the class's name—in this case, `Account`. Every class's body is enclosed in an opening *left brace* (end of line 6) and a closing *right brace* (line 19). The class definition terminates with a *required* semicolon (line 19). For reusability, place each class definition in a separate header with the `.h` filename extension (`Account.h` in this example).

**Common Programming Error 3.1**

*Forgetting the semicolon at the end of a class definition is a syntax error.*

#### *Identifiers and Camel-Case Naming*

Class names, member-function names and data-member names are all *identifiers*. By convention, variable-name identifiers begin with a lowercase letter, and every word in the name after the first word begins with a capital letter—e.g., `firstNumber` starts its second word, `Number`, with a capital `N`. This naming convention is known as camel case, because the uppercase letters stand out like a camel's humps. Also by convention, class names begin with an initial uppercase letter, and member-function and data member-names begin with an initial lowercase letter.

### 3.3.3 Data Member name of Type `string`

Recall from Section 1.8 that an object has attributes, implemented as data members. The object carries these with it throughout its lifetime. Each object has its own copy of the class's data members. Normally, a class also contains one or more member functions. These manipulate the data members belonging to particular objects of the class. The data members exist

- *before* a program calls member functions on an object,

- *while* the member functions are executing and
- *after* the member functions complete execution.

Data members are declared *inside* a class definition but *outside* the bodies of the class's member functions. Line 18

```
std::string name; // data member containing account holder's name
```

declares data member `name` of type `string`. If there are many `Account` objects, each has its own `name`. Because `name` is a data member, it can be manipulated by each of the class's member functions. The default value for a `string` is the **empty `string`** (i.e., `""`)—this is why line 13 in `main` (Fig. 3.1) did not display a name the first time we called `myAccount`'s `getName` member function. Section 3.4 explains how a `string` receives its default value.

**Good Programming Practice 3.1**

*By convention, place a class's data members last in the class's body. You can list the class's data members anywhere in the class outside its member-function definitions, but scattering the data members can lead to hard-to-read code.*

### *Use* `std::` *with Standard Library Components in Headers*

Throughout the `Account.h` header (Fig. 3.2), we use `std::` when referring to `string` (lines 9, 14 and 18). For subtle reasons that we explain in Section 23.4, headers should *not* contain `using` directives or `using` declarations.

## 3.3.4 setName Member Function

Let's walk through the code of member function `setName`'s definition (lines 9–11):

```
void setName(std::string accountName) {
   name = accountName; // store the name
}
```

We refer to the first line of each function definition (line 9) as the *function header*. The member function's **return type** (which appears to the left of the function's name) specifies the type of data the member function returns to its *caller* after performing its task. The return type **`void`** (line 9) indicates that when `setName` completes its task, it does *not* return (i.e., give back) any information to its **calling function**—in this example, line 19 of the `main` function (Fig. 3.1). As you'll soon see, `Account` member function `getName` does return a value.

### `setName` *Parameter*

Our car analogy from Section 1.8 mentioned that pressing a car's gas pedal sends a *message* to the car to perform a task—make the car go faster. But *how fast* should the car accelerate? The farther down you press the pedal, the faster the car accelerates. So the message to the car includes *both* the *task to perform* and *information that helps the car perform that task*. This information is known as a **parameter**—the parameter's *value* helps the car determine how fast to accelerate. Similarly, a member function can require one or more parameters that represent the data it needs to perform its task.

Member function `setName` declares the `string` *parameter* `accountName`—which receives the name that's passed to `setName` as an *argument*. When line 19 in Fig. 3.1

```
myAccount.setName(theName); // put theName in myAccount
```

executes, the *argument value* in the call's parentheses (i.e., the value stored in `theName`) is copied into the corresponding *parameter* (`accountName`) in the member function's header (line 9 of Fig. 3.2). In Fig. 3.1's sample execution, we entered `"Jane Green"` for `theName`, so `"Jane Green"` was copied into the `accountName` parameter.

### `setName` *Parameter List*

Parameters like `accountName` are declared in a **parameter list** located in the *required* parentheses following the member function's name. Each parameter *must* specify a type (e.g., `string`) followed by a parameter name (e.g., `accountName`). When there are multiple parameters, each is separated from the next by a comma, as in

```
    (type1 name1, type2 name2, ...)
```

The number and order of *arguments* in a function call *must match* the number and order of *parameters* in the function definition's parameter list.

### `setName` *Member Function Body*

Every *member function body* is delimited by an opening *left brace* (end of line 9 of Fig. 3.2) and a closing *right brace* (line 11). Within the braces are one or more statements that perform the member function's task(s). In this case, the member function body contains a single statement (line 10)

```
    name = accountName; // store the account name
```

that assigns the `accountName` *parameter's* value (a `string`) to the class's `name` *data member*, thus storing the account name in the object for which `setName` was called—`myAccount` in this example's `main` program.[6] After line 10 executes, program execution reaches the member function's closing brace (line 11), so the function returns to its *caller*.

### *Parameters Are Local Variables*

In Chapter 2, we declared all of a program's variables in the `main` function. Variables declared in a particular function's body are **local variables** which can be used *only* in that function. When a function terminates, the values of its local variables are *lost*. A function's parameters also are local variables of that function.

### *Argument and Parameter Types Must Be Consistent*

The argument types in the member function call must be *consistent* with the types of the corresponding parameters in the member function's definition. (As you'll see in Chapter 6, Functions and an Introduction to Recursion, an argument's type and its corresponding parameter's type are *not* required to be identical.) In our example, the member function call passes one argument of type `string` (`theName`)—and the member function definition specifies one parameter of type `string` (`accountName`). So in this example, the type of the argument in the member function call happens to exactly match the type of the parameter in the member function header.

6. We used different names for the `setName` member function's parameter (`accountName`) and the data member (`name`). It's common idiom in industry to use the same name for both. We'll show you how to do this without ambiguity in Chapter 9.

### 3.3.5 getName Member Function

Member function getName (lines 14–16)

```
std::string getName() const {
   return name; // return name's value to this function's caller
}
```

*returns* a particular Account object's name to the caller—a string, as specified by the function's return type. The member function has an empty parameter list, so it does not require additional information to perform its task. When a member function with a return type other than void is called and completes its task, it *must* return a result to its caller. A statement that calls member function getName on an Account object expects to receive the Account's name.

The return statement in line 15

```
return name; // return name's value to this function's caller
```

passes the string value of data member name back to the caller, which then can use the returned value. For example, the statement in lines 22–23 of Fig. 3.1

```
cout << "Name in object myAccount is: "
   << myAccount.getName() << endl;
```

uses the value returned by getName to output the name stored in the myAccount object.

#### *const Member Functions*

We declared member function getName as const in line 14 of Fig. 3.2

```
std::string getName() const {
```

because in the process of returning the name the function *does not*, and *should not*, modify the Account object on which it's called.

**Error-Prevention Tip 3.1**

*Declaring a member function with const to the right of the parameter list tells the compiler, "this function should* not *modify the object on which it's called—if it does, please issue a compilation error." This can help you locate errors if you accidentally insert in the member function code that would modify the object.*

### 3.3.6 Access Specifiers private and public

The keyword private (line 17)

```
private:
```

is an access specifier. Access specifiers are always followed by a colon (:). Data member name's declaration (line 18) appears *after* access specifier private: to indicate that name is accessible *only* to class Account's member functions.[7] This is known as data hiding—the data member name is *encapsulated* (hidden) and can be used *only* in class Account's setName and getName member functions. Most data-member declarations appear after the private: access specifier. For the remainder of the text, when we refer to the access specifiers private and public in the text, we'll often omit the colon as we did in this sentence.

7. Or to "friends" of the class as you'll see in Section 9.13.

This class also contains the **public** access specifier (line 7)

```
public:
```

Data members or member functions listed after access specifier `public` (and *before* the next access specifier if there is one) are "available to the public." They can be used by other functions in the program (such as `main`), and by member functions of other classes (if there are any). In Chapter 11, we'll introduce the `protected` access specifier.

### *Default Access for Class Members*

By default, everything in a class is `private`, unless you specify otherwise. Once you list an access specifier, everything from that point has that access until you list another access specifier. We prefer to list `public` only once, grouping everything that's `public`, and we prefer to list `private` only once, grouping everything that's `private`. The access specifiers `public` and `private` may be repeated, but this is unnecessary and can be confusing.

**Error-Prevention Tip 3.2**

*Making a class's data members `private` and member functions `public` facilitates debugging because problems with data manipulations are localized to the member functions.*

**Common Programming Error 3.2**

*An attempt by a function that's not a member of a particular class to access a `private` member of that class is a compilation error.*

## 3.3.7 Account UML Class Diagram

We'll often use UML class diagrams to summarize a class's *attributes* and *operations*. In industry, UML diagrams help systems designers specify systems in a concise, graphical, programming-language-independent manner, before programmers implement the systems in specific programming languages. Figure 3.3 presents a UML class diagram for class `Account` of Fig. 3.2.

### *Top Compartment*

In the UML, each class is modeled in a class diagram as a rectangle with three compartments. In this diagram the top compartment contains the *class name* `Account` centered horizontally in boldface type.

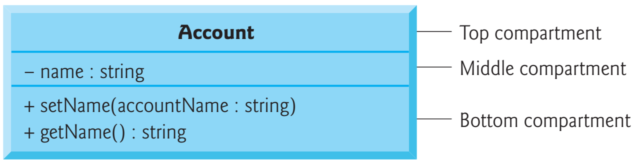

**Fig. 3.3** | UML class diagram for class `Account` of Fig. 3.2.

### *Middle Compartment*

The middle compartment contains the class's attribute `name`, which corresponds to the data member of the same name in C++. Data member `name` is `private` in C++, so the

UML class diagram lists a *minus sign (–) access modifier* before the attribute name. Following the attribute name are a *colon* and the *attribute type*, in this case `string`.

### *Bottom Compartment*

The bottom compartment contains the class's **operations**, `setName` and `getName`, which correspond to the member functions of the same names in C++. The UML models operations by listing the operation name preceded by an *access modifier*, in this case `+ setName`. This plus sign (+) indicates that `setName` is a public operation in the UML (because it's a `public` member function in C++). Operation `getName` is also a public operation.

### *Return Types*

The UML indicates the return type of an operation by placing a colon and the return type after the parentheses following the operation name. `Account` member function `setName` does not return a value (because it returns `void` in C++), so the UML class diagram does not specify a return type after the parentheses of this operation. Member function `getName` has a `string` return type.

### *Parameters*

The UML models a parameter by listing the parameter name, followed by a colon and the parameter type in the parentheses after the operation name. The UML has its own data types similar to those of C++—for simplicity, we use the C++ types. `Account` member function `setName` has a `string` parameter called `accountName`, so the class diagram lists `accountName : string` between the parentheses following the member function name. Operation `getName` does not have any parameters, so the parentheses following the operation name in the class diagram are empty, just as they are in the member function's definition in line 14 of Fig. 3.2.

## 3.4 Account Class: Initializing Objects with Constructors

As mentioned in Section 3.3, when an `Account` object is created, its `string` data member `name` is initialized to the empty `string` by *default*—we'll discuss how that occurs shortly. But what if you want to provide a name when you first *create* an `Account` object? Each class can define a **constructor** that specifies *custom initialization* for objects of that class. A constructor is a special member function that *must* have the *same name* as the class. C++ *requires* a constructor call when *each* object is created, so this is the ideal point to initialize an object's data members.[8]

Like member functions, a constructor can have parameters—the corresponding argument values help initialize the object's data members. For example, you can specify an `Account` object's name when the object is created, as you'll do in line 11 of Fig. 3.5:

```
    Account account1{"Jane Green"};
```

In this case, the `string` argument `"Jane Green"` is passed to the `Account` class's constructor and used to initialize the `name` data member of the `account1` object. The preceding statement assumes that the `Account` class has a constructor that takes only a `string` parameter.

8. In Section 9.6, you'll learn that classes can have multiple constructors.

### 3.4.1 Defining an Account Constructor for Custom Object Initialization

Figure 3.4 shows class `Account` with a constructor that receives an `accountName` parameter and uses it to initialize data member `name` when an `Account` object is created.

```
 1  // Fig. 3.4: Account.h
 2  // Account class with a constructor that initializes the account name.
 3  #include <string>
 4
 5  class Account {
 6  public:
 7     // constructor initializes data member name with parameter accountName
 8     explicit Account(std::string accountName)
 9        : name{accountName} { // member initializer
10       // empty body
11     }
12
13     // function to set the account name
14     void setName(std::string accountName) {
15        name = accountName;
16     }
17
18     // function to retrieve the account name
19     std::string getName() const {
20        return name;
21     }
22  private:
23     std::string name; // account name data member
24  }; // end class Account
```

**Fig. 3.4** | Account class with a constructor that initializes the account name.

***Account Class's Constructor Definition***

Lines 8–11 of Fig. 3.4

```
   explicit Account(std::string accountName)
      : name{accountName} { // member initializer
     // empty body
   }
```

define `Account`'s constructor. Normally, constructors are `public`.[9]

A constructor's *parameter list* specifies pieces of data required to initialize an object. Line 8

```
   explicit Account(std::string accountName)
```

indicates that the constructor has one `string` parameter called `accountName`. When you create a new `Account` object, you *must* pass a person's name to the constructor, which will receive that name in the *parameter* `accountName`. The constructor will then use `accountName` to initialize the *data member* `name`.

9. Section 10.10.2 discusses why you might use a `private` constructor.

The constructor uses a **member-initializer list** (line 9)

```
      : name{accountName}
```

to initialize the `name` data member with the value of the parameter `accountName`. *Member initializers* appear between a constructor's parameter list and the left brace that begins the constructor's body. The member initializer list is separated from the parameter list with a colon (`:`). Each member initializer consists of a data member's *variable name* followed by parentheses containing the member's *initial value*. In this example, `name` is initialized with the parameter `accountName`'s value. If a class contains more than one data member, each member initializer is separated from the next by a comma. The member initializer list executes *before* the constructor's body executes.

**Performance Tip 3.1**

*You can perform initialization in the constructor's body, but you'll learn in Chapter 9 that it's more efficient to do it with member initializers, and some types of data members must be initialized this way.*

### *`explicit` Keyword*

We declared this constructor **`explicit`**, because it takes a *single* parameter—this is important for subtle reasons that you'll learn in Section 10.13. For now, just declare *all* single-parameter constructors `explicit`. Line 8 of Fig. 3.4 does *not* specify a return type, because constructors *cannot* return values—not even `void`. Also, constructors cannot be declared `const` (because initializing an object modifies it).

### *Using the Same Parameter Name in the Constructor and Member Function `setName`*

Recall from Section 3.3.4 that member function parameters are local variables. In Fig. 3.4, the constructor and member function `setName` both have a parameter called `accountName`. Though their identifiers are identical, the parameter in line 8 is a local variable of the constructor that's *not* visible to member function `setName`. Similarly, the parameter in line 14 is a local variable of `setName` that's *not* visible to the constructor. Such visibility is called *scope*, which is discussed in Section 6.10.

## 3.4.2 Initializing Account Objects When They're Created

The `AccountTest` program (Fig. 3.5) initializes two different `Account` objects using the constructor. Line 11

```
      Account account1{"Jane Green"};
```

creates the `Account` object `account1`. When you create an object, C++ implicitly calls the class's constructor to *initialize* that object. If the constructor has parameters, you place the corresponding arguments in braces, `{` and `}`, to the right of the object's variable name. In line 11, the *argument* `"Jane Green"` initializes the new object's `name` data member. Line 12

```
      Account account2{"John Blue"};
```

repeats this process, passing the argument `"John Blue"` to initialize `name` for `account2`. Lines 15–16 use each object's `getName` member function to obtain the names and show that they were indeed initialized when the objects were created. The output shows *different* names, confirming that each `Account` maintains its *own copy* of data member `name`.

```
// Fig. 3.5: AccountTest.cpp
// Using the Account constructor to initialize the name data
// member at the time each Account object is created.
#include <iostream>
#include "Account.h"

using namespace std;

int main() {
   // create two Account objects
   Account account1{"Jane Green"};
   Account account2{"John Blue"};

   // display initial value of name for each Account
   cout << "account1 name is: " << account1.getName() << endl;
   cout << "account2 name is: " << account2.getName() << endl;
}
```

```
account1 name is: Jane Green
account2 name is: John Blue
```

**Fig. 3.5** | Using the Account constructor to initialize the name data member at the time each Account object is created.

### *Default Constructor*

Recall that line 10 of Fig. 3.1

```
    Account myAccount;
```

creates an Account object *without* placing braces to the right of the object's variable name. In this case, C++ implicitly calls the class's **default constructor**. In any class that does *not* explicitly define a constructor, the compiler provides a default constructor with no parameters. The default constructor does *not* initialize the class's fundamental-type data members, but *does* call the default constructor for each data member that's an object of another class. For example, in the Account class of Fig. 3.2, the class's default constructor calls class string's default constructor to initialize the data member name to the empty string. An uninitialized fundamental-type variable contains an undefined ("garbage") value.[10]

### *There's No Default Constructor in a Class That Defines a Constructor*

If you define a custom constructor for a class, the compiler will *not* create a default constructor for that class. In that case, you will not be able to create an Account object using

```
    Account myAccount;
```

as we did in Fig. 3.1, unless the custom constructor you define has an empty parameter list. We'll show later that C++11 allows you to force the compiler to create the default constructor even if you've defined non-default constructors. 11

10. We'll see an exception to this in Section 6.10.

**Software Engineering Observation 3.1**

*Unless default initialization of your class's data members is acceptable, you should generally provide a custom constructor to ensure that your data members are properly initialized with meaningful values when each new object of your class is created.*

### 3.4.3 Account UML Class Diagram with a Constructor

The UML class diagram of Fig. 3.6 models class `Account` of Fig. 3.4, which has a constructor with a `string accountName` parameter. Like operations (Fig. 3.3), the UML models constructors in the *third* compartment of a class diagram. To distinguish a constructor from the class's operations, the UML requires that the word "constructor" be enclosed in **guillemets (« and »)** and placed before the constructor's name. It's customary to list constructors *before* other operations in the third compartment.

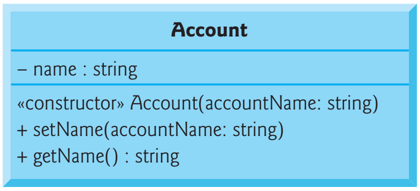
Account
– name : string
«constructor» Account(accountName: string)
+ setName(accountName: string)
+ getName() : string

**Fig. 3.6** | UML class diagram for the `Account` class of Fig. 3.4.

## 3.5 Software Engineering with *Set* and *Get* Member Functions

As you'll see in the next section, *set* and *get* member functions can *validate* attempts to modify `private` data and control how that data is presented to the caller, respectively. These are compelling software engineering benefits.

If a data member were `public`, any **client** of the class—that is, any other code that calls the class's member functions—could see the data and do whatever it wanted with it, including setting it to an *invalid* value.

You might think that even though a client of the class cannot directly access a `private` data member, the client can nevertheless do whatever it wants with the variable through `public` *set* and *get* functions. You'd think that you could peek at the `private` data (and see exactly how it's stored in the object) any time with the `public` *get* function and that you could modify the `private` data at will through the `public` *set* function.

Actually, *set* functions can be programmed to *validate* their arguments and reject any attempts to *set* the data to bad values, such as

- a negative body temperature
- a day in March outside the range 1 through 31
- a product code not in the company's product catalog, etc.

And a *get* function can present the data in a different form, while the actual data representation remains hidden from the user. For example, a `Grade` class might store a `grade` data member as an `int` between 0 and 100, but a `getGrade` member function might return a letter grade as a `string`, such as `"A"` for grades between 90 and 100, `"B"` for grades between 80 and 89, etc. Tightly controlling the *access* to and *presentation* of `private` data can greatly reduce errors, while increasing the robustness, security and usability of your programs.

### *Conceptual View of an* `Account` *Object with Encapsulated Data*

You can think of an `Account` object as shown in Fig. 3.7. The `private` data member `name` is *hidden inside* the object (represented by the inner circle containing `name`) and *protected by an outer layer* of `public` member functions (represented by the outer circle containing `getName` and `setName`). Any client code that needs to interact with the `Account` object can do so *only* by calling the `public` member functions of the protective outer layer.

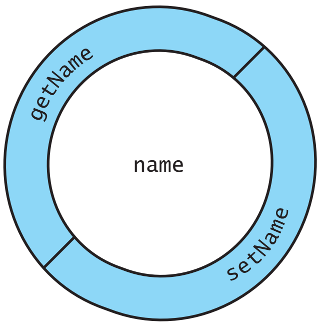

**Fig. 3.7** | Conceptual view of an `Account` object with its encapsulated `private` data member `name` and protective layer of `public` member functions.

**Software Engineering Observation 3.2**

*Generally, data members should be* `private` *and member functions* `public`*. In Chapter 9, we'll discuss why you might use a* `public` *data member or a* `private` *member function.*

**Software Engineering Observation 3.3**

*Using* `public` set *and* get *functions to control access to* `private` *data makes programs clearer and easier to maintain. Change is the rule rather than the exception. You should anticipate that your code will be modified, and possibly often.*

## 3.6 Account Class with a Balance; Data Validation

We now define an `Account` class that maintains a bank account's `balance` in addition to the `name`. In Chapter 2 we used the data type `int` to represent integers. For simplicity, we'll use data type `int` to represent the account balance. In Chapter 4, you'll see how to represent numbers with *decimal points*.

### 3.6.1 Data Member `balance`

A typical bank services many accounts, each with its own balance. In this updated `Account` class (Fig. 3.8), line 42

```
    int balance{0}; // data member with default initial value
```

declares a data member `balance` of type `int` and initializes its value to `0`. This is known as
11 an **in-class initializer** and was introduced in C++11. Every object of class `Account` contains
its *own* copies of *both* the `name` and the `balance`.

```cpp
// Fig. 3.8: Account.h
// Account class with name and balance data members, and a
// constructor and deposit function that each perform validation.
#include <string>

class Account {
public:
   // Account constructor with two parameters
   Account(std::string accountName, int initialBalance)
      : name{accountName} { // assign accountName to data member name

      // validate that the initialBalance is greater than 0; if not,
      // data member balance keeps its default initial value of 0
      if (initialBalance > 0) { // if the initialBalance is valid
         balance = initialBalance; // assign it to data member balance
      }
   }

   // function that deposits (adds) only a valid amount to the balance
   void deposit(int depositAmount) {
      if (depositAmount > 0) { // if the depositAmount is valid
         balance = balance + depositAmount; // add it to the balance
      }
   }

   // function returns the account balance
   int getBalance() const {
      return balance;
   }

   // function that sets the name
   void setName(std::string accountName) {
      name = accountName;
   }

   // function that returns the name
   std::string getName() const {
      return name;
   }
private:
   std::string name; // account name data member
   int balance{0}; // data member with default initial value
}; // end class Account
```

**Fig. 3.8** | `Account` class with `name` and `balance` data members, and a constructor and `deposit` function that each perform validation.

***Account's Member Functions Can All Use balance***

The statements in lines 15, 22 and 28 use the variable `balance` even though it was *not* declared in *any* of the member functions. We can use `balance` in these member functions because it's a *data member* in the same class definition.

## 3.6.2 Two-Parameter Constructor with Validation

The class has a *constructor* and four *member functions*. It's common for someone opening an account to deposit money immediately, so the constructor (lines 9–17) now receives a second parameter—`initialBalance` of type `int` that represents the *starting balance*. We did not declare this constructor `explicit` (as in Fig. 3.4), because this constructor has more than one parameter.

Lines 14–16 of Fig. 3.8

```
if (initialBalance > 0) { // if the initialBalance is valid
   balance = initialBalance; // assign it to data member balance
}
```

ensure that data member `balance` is assigned parameter `initialBalance`'s value *only* if that value is greater than `0`—this is known as validation or validity checking. If so, line 15 assigns `initialBalance`'s value to data member `balance`. Otherwise, `balance` remains at `0`—its *default initial value* that was set at line 42 in class `Account`'s definition.

## 3.6.3 deposit Member Function with Validation

Member function `deposit` (lines 20–24) does *not* return any data when it completes its task, so its return type is `void`. The member function receives one `int` parameter named `depositAmount`. Lines 21–23

```
if (depositAmount > 0) { // if the depositAmount is valid
   balance = balance + depositAmount; // add it to the balance
}
```

ensure that parameter `depositAmount`'s value is added to the `balance` only if the parameter value is valid (i.e., greater than zero)—another example of validity checking. Line 22 first adds the current `balance` and `depositAmount`, forming a *temporary* sum which is then assigned to `balance`, *replacing* its prior value (recall that addition has a higher precedence than assignment). It's important to understand that the calculation

```
balance + depositAmount
```

on the right side of the assignment operator in line 22 does *not* modify the balance—that's why the assignment is necessary. Section 4.12 shows a more concise way to write line 22.

## 3.6.4 getBalance Member Function

Member function `getBalance` (lines 27–29) allows the class's *clients* to obtain the value of a particular `Account` object's `balance`. The member function specifies return type `int` and an *empty* parameter list. Like member function `getName`, `getBalance` is declared `const`, because in the process of returning the `balance` the function does not, and should not, modify the `Account` object on which it's called.

## 3.6.5 Manipulating Account Objects with Balances

The `main` function in Fig. 3.9 creates two `Account` objects (lines 10–11) and attempts to initialize them with a *valid* balance of `50` and an *invalid* balance of `-7`, respectively—for the purpose of our examples, we assume that balances must be greater than or equal to zero. Lines 14–17 output the account names and balances, which are obtained by calling each `Account`'s `getName` and `getBalance` member functions.

```
 1  // Fig. 3.9: AccountTest.cpp
 2  // Displaying and updating Account balances.
 3  #include <iostream>
 4  #include "Account.h"
 5
 6  using namespace std;
 7
 8  int main()
 9  {
10     Account account1{"Jane Green", 50};
11     Account account2{"John Blue", -7};
12
13     // display initial balance of each object
14     cout << "account1: " << account1.getName() << " balance is $"
15        << account1.getBalance();
16     cout << "\naccount2: " << account2.getName() << " balance is $"
17        << account2.getBalance();
18
19     cout << "\n\nEnter deposit amount for account1: "; // prompt
20     int depositAmount;
21     cin >> depositAmount; // obtain user input
22     cout << "adding " << depositAmount << " to account1 balance";
23     account1.deposit(depositAmount); // add to account1's balance
24
25     // display balances
26     cout << "\n\naccount1: " << account1.getName() << " balance is $"
27        << account1.getBalance();
28     cout << "\naccount2: " << account2.getName() << " balance is $"
29        << account2.getBalance();
30
31     cout << "\n\nEnter deposit amount for account2: "; // prompt
32     cin >> depositAmount; // obtain user input
33     cout << "adding " << depositAmount << " to account2 balance";
34     account2.deposit(depositAmount); // add to account2 balance
35
36     // display balances
37     cout << "\n\naccount1: " << account1.getName() << " balance is $"
38        << account1.getBalance();
39     cout << "\naccount2: " << account2.getName() << " balance is $"
40        << account2.getBalance() << endl;
41  }
```

**Fig. 3.9** | Displaying and updating `Account` balances. (Part 1 of 2.)

```
account1: Jane Green balance is $50
account2: John Blue balance is $0

Enter deposit amount for account1: 25
adding 25 to account1 balance

account1: Jane Green balance is $75
account2: John Blue balance is $0

Enter deposit amount for account2: 123
adding 123 to account2 balance

account1: Jane Green balance is $75
account2: John Blue balance is $123
```

**Fig. 3.9** | Displaying and updating `Account` balances. (Part 2 of 2.)

### *Displaying the* `Account` *Objects' Initial Balances*

When member function `getBalance` is called for `account1` from line 15, the value of `account1`'s `balance` is returned from line 28 of Fig. 3.8 and displayed by the output statement in lines 14–15 (Fig. 3.9). Similarly, when member function `getBalance` is called for `account2` from line 17, the value of the `account2`'s `balance` is returned from line 28 of Fig. 3.8 and displayed by the output statement (Fig. 3.9, lines 16–17). The `balance` of `account2` is initially `0`, because the constructor rejected the attempt to start `account2` with a negative balance, so the data member `balance` retains its default initial value.

### *Reading a Deposit Amount from the User and Making a Deposit*

Line 19 prompts the user to enter a deposit amount for `account1`. Line 20 declares local variable `depositAmount` to store each deposit amount entered by the user. We did not initialize `depositAmount`, because as you'll learn momentarily, variable `depositAmount`'s value will be input by the user's input.

**Error-Prevention Tip 3.3**

*Most C++ compilers issue a warning if you attempt to use the value of an uninitialized variable. This helps you avoid dangerous execution-time logic errors. It's always better to get the warnings and errors out of your programs at compilation time rather than execution time.*

Line 21 reads the deposit amount from the user and places the value into local variable `depositAmount`. Line 22 displays the deposit amount. Line 23 calls object `account1`'s `deposit` member function with the `depositAmount` as the member function's *argument*. When the member function is called, the argument's value is assigned to the parameter `depositAmount` of member function `deposit` (line 20 of Fig. 3.8); then member function `deposit` adds that value to the `balance`. Lines 26–29 (Fig. 3.9) output the `name`s and `balance`s of both `Account`s again to show that *only* `account1`'s `balance` has changed.

Line 31 prompts the user to enter a deposit amount for `account2`. Line 32 obtains the input from the user. Line 33 displays the `depositAmount`. Line 34 calls object `account2`'s `deposit` member function with `depositAmount` as the member function's *argument*; then member function `deposit` adds that value to the `balance`. Finally, lines 37–40 output the `name`s and `balance`s of both `Account`s again to show that *only* `account2`'s `balance` has changed.

### *Duplicated Code in the* `main` *Function*

The six statements at lines 14–15, 16–17, 26–27, 28–29, 37–38 and 39–40 are almost identical. Each outputs an `Account`'s `name` and `balance`, and differs only in the `Account` object's name—`account1` or `account2`. Duplicate code like this can create *code maintenance problems* when that code needs to be updated. For example, if *six* copies of the same code all have the same error to fix or the same update to be made, you must make that change six times, without making errors. Exercise 3.13 asks you to modify Fig. 3.9 to include function `displayAccount` that takes as a parameter an `Account` object and outputs the object's `name` and `balance`. You'll then replace `main`'s duplicated statements with six calls to `displayAccount`.

**Software Engineering Observation 3.4**

*Replacing duplicated code with calls to a function that contains only one copy of that code can reduce the size of your program and improve its maintainability.*

## 3.6.6 Account UML Class Diagram with a Balance and Member Functions `deposit` and `getBalance`

The UML class diagram in Fig. 3.10 concisely models class `Account` of Fig. 3.8. The diagram models in its second compartment the `private` attributes `name` of type `string` and `balance` of type `int`.

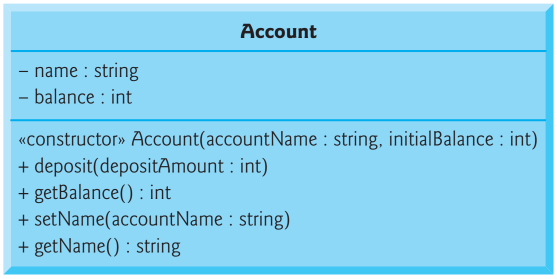
Account
– name : string
– balance : int
«constructor» Account(accountName : string, initialBalance : int)
+ deposit(depositAmount : int)
+ getBalance() : int
+ setName(accountName : string)
+ getName() : string

**Fig. 3.10** | UML class diagram for the `Account` class of Fig. 3.8.

Class `Account`'s constructor is modeled in the third compartment with parameters `accountName` of type `string` and `initialBalance` of type `int`. The class's four `public` member functions also are modeled in the third compartment—operation `deposit` with a `depositAmount` parameter of type `int`, operation `getBalance` with a return type of `int`, operation `setName` with an `accountName` parameter of type `string` and operation `getName` with a return type of `string`.

## 3.7 Wrap-Up

In this chapter, you created your own classes and member functions, created objects of those classes and called member functions of those objects to perform useful actions. You declared data members of a class to maintain data for each object of the class, and you defined your own member functions to operate on that data. You passed information to a member function as arguments whose values are assigned to the member function's parameters. You learned the difference between a local variable of a member function and a data member of a class, and that only data members that are objects are initialized automatically with calls to their default constructors. You also learned how to use a class's constructor to specify the initial values for an object's data members. You saw how to create UML class diagrams that model the member functions, attributes and constructors of classes.

In the next chapter we begin our introduction to control statements, which specify the order in which a program's actions are performed. You'll use these in your member functions to specify how they should order their tasks.

## Summary

### *Section 3.1 Introduction*

- Each class you create becomes a new type you can use to declare variables and create objects.
- C++ is an **extensible programming language** (p. 114)—you can define new class types as needed.

### *Section 3.2 Test-Driving an `Account` Object*

- Classes cannot execute by themselves.
- A `main` function can "drive" an object by calling its member functions—without knowing how the class is implemented. In this sense, `main` is referred to as a **driver program** (p. 115).

### *Section 3.2.1 Instantiating an Object*

- Typically, you cannot call a member function of a class until you create an object of that class.

### *Section 3.2.2 Headers and Source-Code Files*

- The compiler knows about fundamental types that are "built into" C++.
- A new type that you create is known as a **user-defined type** (p. 116).
- New classes, when packaged properly, can be reused by other programmers.
- Reusable code (such as a class definition) is placed in a file known as a **header** (p. 116) that you include (via `#include`) wherever you need to use the code.
- By convention, a header for a user-defined type has a `.h` filename extension.
- In an `#include` directive, a user-defined header is placed in double quotes (`""`), indicating that the header is located with your program, rather than with the C++ Standard Library headers.
- Files ending in `.cpp` are known as **source-code files** (p. 116).

### *Section 3.2.3 Calling Class `Account`'s `getName` Member Function*

- To call a member function for a specific object, you specify the object's name, followed by a **dot operator** (`.`; p. 116), then the member function name and a set of parentheses. *Empty* parentheses indicate that the function does not require any additional information to perform its task.
- A member function can return a value from the object on which the function is called.

### *Section 3.2.4 Inputting a `string` with `getline`*

- Functions that are not members of a class are called **global functions** (p. 117).
- An object of C++ Standard Library class **`string`** (p. 117) stores character string values. Class `string` is defined in the **`<string>` header** (p. 117) and belongs to namespace `std`.

- C++ Standard Library function **getline** (p. 117), from the <string> header, reads characters up to, but not including, a newline, which is discarded, then places the characters in a string.

### *Section 3.2.5 Calling Class* **Account***'s* **setName** *Member Function*

- A member-function call can supply **arguments** (p. 118) that help the function perform its task.

### *Section 3.3.1* **Account** *Class Definition*

- A class's data members maintain data for each object of the class, and its member functions manipulate the class's data members.

### *Section 3.3.2 Keyword* **class** *and the Class Body*

- A class definition begins with keyword **class** (p. 119) followed immediately by the class's name.
- A class's body is enclosed in an opening left brace and a closing right brace.
- A class definition terminates with a required semicolon.
- Typically, each class definition is placed in a separate header with the .h filename extension.
- Class names, member function names and data member names are all identifiers. By convention, variable-name identifiers begin with a lowercase letter, and every word in the name after the first word begins with a capital letter. This naming convention is known as camel case, because the uppercase letters stand out like a camel's humps. Also by convention, class names begin with an initial uppercase letter, and member function and data member names begin with an initial lowercase letter.

### *Section 3.3.3 Data Member* **name** *of Type* **string**

- Each object of a class has its own copy of the class's data members.
- An object's data members exist before a program calls member functions on an object, while they are executing and after the member functions complete execution.
- Data members are declared inside a class definition but outside its member functions' bodies.
- The default value for a string is the **empty string** (i.e., ""; p. 120).
- Headers should never contain using directives or using declarations.

### *Section 3.3.4* **setName** *Member Function*

- A function's **return type** (p. 119; which appears to the left of the function's name) specifies the type of data the function returns to its caller after performing its task.
- The return type **void** (p. 120) indicates that when a function completes its task, it does not return (i.e., give back) any information to its **calling function** (p. 120).
- **Parameters** (p. 120) specify additional information the function needs to perform its task.
- When you call a function, each argument value in the call's parentheses is copied into the corresponding parameter in the member function definition.
- Parameters are declared in a **parameter list** (p. 121) located in required parentheses following a function's name. Each parameter must specify a type followed by a parameter name.
- Multiple parameters in a function definition are separated by commas.
- The number and order of arguments in a function call must match the number and order of parameters in the function definition's parameter list.
- Every function body is delimited by an opening left brace and a closing right brace. Within the braces are one or more statements that perform the function's task(s).
- When program execution reaches a function's closing brace, the function returns to its caller.

- Variables declared in a particular function's body are **local variables** (p. 121), which can be used only in that function. When a function terminates, the values of its local variables are lost.
- A function's parameters also are local variables of that function.
- The argument types in the member function call must be consistent with the types of the corresponding parameters in the member function's definition.

### *Section 3.3.5* **`getName`** *Member Function*

- When a member function that specifies a return type other than `void` is called and completes its task, it must return a result to its caller.
- The **`return`** statement (p. 122) passes a value back to a function's caller.
- A member function that does not, and should not, modify the object on which it's called is declared with **`const`** (p. 122) to the right of its parameter list.

### *Section 3.3.6 Access Specifiers* **`private`** *and* **`public`**

- The keyword **`private`** (p. 122) is an **access specifier** (p. 122).
- Access specifiers are always followed by a colon (`:`).
- A `private` data member is accessible only to its class's member functions.
- Most data-member declarations appear after the `private` access specifier.
- Variables or functions listed after the **`public`** (p. 123) access specifier (and before the next access specifier, if there is one) are "available to the public." They can be used by other functions in the program, and by member functions of other classes.
- By default, everything in a class is `private`, unless you specify otherwise.
- Once you list an access specifier, everything from that point has that access until you list another access specifier.
- Declaring data members `private` is known as **data hiding** (p. 122). `private` data members are encapsulated (hidden) in an object and can be accessed only by member functions of the object's class.

### *Section 3.3.7* **`Account`** *UML Class Diagram*

- **UML class diagrams** (p. 123) can be used to summarize a class's attributes and operations.
- In the UML, each class is modeled in a class diagram as a rectangle with three compartments.
- The top compartment contains the class name centered horizontally in boldface type.
- The middle compartment contains the class's attribute names, which correspond to the data members of a class.
- A private attribute lists a minus sign (–) access modifier before the attribute name.
- Following the attribute name are a colon and the attribute type.
- The bottom compartment contains the class's **operations** (p. 124), which correspond to the member functions in a class.
- The UML models operations by listing the operation name preceded by an access modifier. A plus sign (+) indicates a public operation in the UML.
- An operation that does not have any parameters specifies empty parentheses following the operation name.
- The UML indicates the return type of an operation by placing a colon and the return type after the parentheses following the operation name.
- For a `void` return type a UML class diagram does not specify anything after the parentheses of the operation.

- The UML models a parameter by listing the parameter name, followed by a colon and the parameter type in the parentheses after the operation name.

### *Section 3.4 `Account` Class: Initializing Objects with Constructors*

- Each class can define a **constructor** (p. 124) for custom object initialization.
- A constructor is a special member function that must have the same name as the class.
- C++ requires a constructor call for every object that's created.
- Like member functions, a constructor can specify parameters—the corresponding argument values help initialize the object's data members.

### *Section 3.4.1 Declaring an `Account` Constructor for Custom Object Initialization*

- Normally, constructors are `public`.
- A constructor's parameter list specifies pieces of data required to initialize an object.
- A constructor uses a **member-initializer list** (p. 126) to initialize its data members with the values of the corresponding parameters.
- Member initializers appear between a constructor's parameter list and the left brace that begins the constructor's body.
- The member-initializer list is separated from the parameter list with a colon (`:`).
- Each member initializer consists of a data member's variable name followed by parentheses containing the member's initial value.
- Each member initializer in a constructor is separated from the next by a comma.
- The member initializer list executes before the constructor's body executes.
- A constructor that specifies a single parameter should be declared **`explicit`** (p. 126).
- A constructor does not specify a return type, because constructors cannot return values.
- Constructors cannot be declared `const` (because initializing an object modifies it).

### *Section 3.4.2 Initializing `Account` Objects When They're Created*

- When you create an object, C++ calls the class's constructor to initialize that object. If a constructor has parameters, the corresponding arguments are placed in braces, `{` and `}`, to the right of the object's variable name.
- When you create an object without placing braces to the right of the object's variable name, C++ implicitly calls the class's **default constructor** (p. 127).
- In any class that does not explicitly define a constructor, the compiler provides a default constructor (which always has no parameters).
- The default constructor does not initialize the class's fundamental-type data members, but does call the default constructor for each data member that's an object of another class.
- A `string`'s default constructor initializes the object to the empty `string`.
- An uninitialized fundamental-type variable contains an undefined ("garbage") value.
- If a class defines a constructor, the compiler will not create a default constructor for that class.

### *Section 3.4.3 `Account` UML Class Diagram with a Constructor*

- Like operations, the UML models constructors in the third compartment of a class diagram.
- To distinguish a constructor from the class's operations, the UML requires that the word "constructor" be enclosed in **guillemets** (**« and »**; p. 128) and placed before the constructor's name.
- It's customary to list constructors before other operations in the third compartment.

### *Section 3.5 Software Engineering with Set and Get Member Functions*

- Through the use of *set* and *get* member functions, you can validate attempted modifications to `private` data and control how that data is presented to the caller.
- A **client** (p. 128) of a class is any other code that calls the class's member functions.
- Any client code can see a `public` data member and do whatever it wanted with it, including setting it to an invalid value.
- *Set* functions can be programmed to validate their arguments and reject any attempts to *set* the data to bad values.
- A *get* function can present the data to a client in a different form.
- Tightly controlling the access to and presentation of `private` data can greatly reduce errors, while increasing the usability, robustness and security of your programs.

### *Section 3.6.1 Data Member* `balance`

- You can initialize fundamental-type data members in their declarations. This is known as an **in-class initializer** (p. 130) and was introduced in C++11.

### *Section 3.6.2 Two-Parameter Constructor with Validation*

- A constructor can perform **validation** (p. 131) or **validity checking** (p. 131) before modifying a data member.

### *Section 3.6.3* `deposit` *Member Function with Validation*

- A *set* function can perform validity checking before modifying a data member.

## Self-Review Exercises

**3.1** Fill in the blanks in each of the following:

a) Every class definition contains the keyword ________ followed immediately by the class's name.
b) Class names, member-function names and data-member names are all ________.
c) Each parameter in a function header specifies both a(n) ________ and a(n) ________.
d) Classes are *executed* by an instance of a class known as a(n) ________.
e) Access specifiers are always followed by a ________.
f) The `private` access specifier is the ________ access specifier unless another is specified.
g) When C++ implicitly calls a function of the class, then that function is known as the class's ________.
h) Any file that uses a class can include the class's header via a(n) ________ preprocessing directive.

**3.2** State whether each of the following is *true* or *false*. If *false*, explain why.

a) You can call a member function of a class even if *an object* of that class has not been created.
b) Empty parentheses following a function name in a function definition indicate that the function does not require any parameters to perform its task.
c) If you define a custom constructor for a class, the compiler will *not* create a default constructor for that class.
d) Variables declared in the body of a particular member function are known as data members and can be used in all member functions of the class.
e) A header that you define in your program is placed in angle brackets (<>).

f) A constructor is a special member function that must have the same name as the class.

**3.3** What is the difference between a local variable and a data member?

**3.4** Explain the purpose of a function parameter. What's the difference between a parameter and an argument?

## Answers to Self-Review Exercises

**3.1** a) `class`. b) identifiers. c) type, name. d) object. e) colon. f) default. g) default constructor. h) `#include`.

**3.2** a) False, a member function of a class can only be called by an object of that class. b) True. c) True. d) False. Such variables are local variables and can be used only in the member function in which they're declared. e) False, it is placed in double quotes (""). f) True.

**3.3** A local variable is declared in the body of a function and can be used only from its declaration to the closing brace of the block in which it's declared. A data member is declared in a class, but not in the body of any of the class's member functions. Every object of a class has each of the class's data members. Data members are accessible to all member functions of the class.

**3.4** A parameter represents additional information that a function requires to perform its task. Each parameter required by a function is specified in the function header. An argument is the value supplied in the function call. When the function is called, the argument value is passed into the function parameter so that the function can perform its task.

## Exercises

**3.5** *(Private Access Specifier)* What are the different uses of private access specifiers?

**3.6** *(Data members)* What is the difference between `private` and `public` data members?

**3.7** *(Using a Class Without a* `using` *Directive)* Explain how a program could use class `string` without inserting a `using` directive.

**3.8** *(*Set *and* Get *Functions)* Explain why a class might provide a *set* function and a *get* function for a data member.

**3.9** *(Modified* `Account` *Class)* Modify class `Account` (Fig. 3.8) to provide a member function called `withdraw` that withdraws money from an `Account`. Ensure that the withdrawal amount does not exceed the `Account`'s balance. If it does, the balance should be left unchanged and the member function should display a message indicating `"Withdrawal amount exceeded account balance."` Modify class `AccountTest` (Fig. 3.9) to test member function `withdraw`.

**3.10** *(Invoice class)* Create a class called `Invoice` that a hardware store might use to represent an invoice for an item sold at the store. An `Invoice` should include six data members-a part number (type `string`), a part description (type `string`), a quantity of the item being purchased (type `int`), a price per item (type `int`) a value-added tax (VAT) rate as a decimal (type `double`) and a discount rate as a decimal(type `double`). Your class should have a constructor that initializes the six data members. The constructor should initialize the first four data members with values from parameters and the last two data members to default values of 0.20 per cent and zero respectively. Provide a *set* and a *get* functions for each data member. In addition, provide a member function named `getInvoiceAmount` that calculates the invoice amount (i.e., multiplies the quantity by the price per item and applies the tax and discount amounts), then returns the amount. Have the set data members perform validity checks on their parameters—if a parameter value is not positive, it should be left unchanged. Write a driver program to demonstrate `Invoice`'s capabilities.

**3.11** *(MotorVehicle class)* Create a class called `MotorVehicle` that represents a motor vehicle using: `make` (type `string`), `fuelType` (type `string`), `yearOfManufacture` (type `int`), `color` (type `string`) and `engineCapacity` (type `int`). Your class should have a constructor that initializes the three data members. Provide a *set* and a *get* function for each data member. Add a member function called `displayCarDetails` that displays the five data members in five separate lines in the form `"member name: member value"`. Write a test program to demonstrate `MotorVehicle`'s capabilities.

**3.12** ***(`Date` Class)*** Create a class called `Date` that includes three pieces of information as data members—a month (type `int`), a day (type `int`) and a year (type `int`). Your class should have a constructor with three parameters that uses the parameters to initialize the three data members. For the purpose of this exercise, assume that the values provided for the year and day are correct, but ensure that the month value is in the range 1–12; if it isn't, set the month to 1. Provide a *set* and a *get* function for each data member. Provide a member function `displayDate` that displays the month, day and year separated by forward slashes (`/`). Write a test program that demonstrates class `Date`'s capabilities.

**3.13** ***(Removing Duplicated Code in the `main` Function)*** In Fig. 3.9, the `main` function contains six statements (lines 14–15, 16–17, 26–27, 28–29, 37–38 and 39–40) that each display an `Account` object's `name` and `balance`. Study these statements and you'll notice that they differ only in the `Account` object being manipulated—`account1` or `account2`. In this exercise, you'll define a new `displayAccount` function that contains *one* copy of that output statement. The member function's parameter will be an `Account` object and the member function will output the object's `name` and `balance`. You'll then replace the six duplicated statements in `main` with calls to `displayAccount`, passing as an argument the specific `Account` object to output.

Modify Fig. 3.9 to define the following `displayAccount` function *after* the `using` directive and *before* `main`:

```
void displayAccount(Account accountToDisplay) {
   // place the statement that displays
   // accountToDisplay's name and balance here
}
```

Replace the comment in the member function's body with a statement that displays `accountToDisplay`'s `name` and `balance`.

Once you've completed `displayAccount`'s declaration, modify `main` to replace the statements that display each `Account`'s `name` and `balance` with calls to `displayAccount` of the form:

```
displayAccount(nameOfAccountObject);
```

In each call, the argument should be the `account1` or `account2` object, as appropriate. Then, test the updated program to ensure that it produces the same output as shown in Fig. 3.9.

**3.14** ***(C++11 List Initializers)*** Write a statement that uses list initialization to initialize an object of class `Account` which provides a constructor that receives an `unsigned int`, two `string`s and a `double` to initialize the `accountNumber`, `firstName`, `lastName` and `balance` data members of a new object of the class.

## Making a Difference

**3.15** ***(Target-Heart-Rate Calculator)*** While exercising, you can use a heart-rate monitor to see that your heart rate stays within a safe range suggested by your trainers and doctors. According to the American Heart Association (AHA) (`http://bit.ly/AHATargetHeartRates`), the formula for calculating your *maximum heart rate* in beats per minute is 220 minus your age in years. Your *target heart rate* is a range that's 50–85% of your maximum heart rate. [*Note:* These formulas are estimates provided by the AHA. Maximum and target heart rates may vary based on the health, fitness and gender of the individual. **Always consult a physician or qualified health-care professional before beginning or modifying an exercise program.**] Create a class called `HeartRates`. The class attributes should include the person's first name, last name and date of birth (consisting of separate attributes for the month, day and year of birth). Your class should have a constructor that receives this data as parameters. For each attribute provide *set* and *get* functions. The class also should include a member function that calculates and returns the person's age (in years), a member function that calculates and returns the person's maximum heart rate and a member function that calculates and returns the person's target heart rate. Write a program that prompts for the person's information, instantiates an object of class `HeartRates` and prints the information from that object—including the person's first name, last name and date of birth—then calculates and prints the person's age in (years), maximum heart rate and target-heart-rate range.

**3.16** *(Computerization of Health Records)* A health-care issue that has been in the news lately is the computerization of health records. This possibility is being approached cautiously because of sensitive privacy and security concerns, among others. [We address such concerns in later exercises.] Computerizing health records could make it easier for patients to share their health profiles and histories among their various health-care professionals. This could improve the quality of health care, help avoid drug conflicts and erroneous drug prescriptions, reduce costs and, in emergencies, could save lives. In this exercise, you'll design a "starter" `HealthProfile` class for a person. The class attributes should include the person's first name, last name, gender, date of birth (consisting of separate attributes for the month, day and year of birth), height (in inches) and weight (in pounds). Your class should have a constructor that receives this data. For each attribute, provide *set* and *get* functions. The class also should include member functions that calculate and return the user's age in years, maximum heart rate and target-heart-rate range (see Exercise 3.15), and body mass index (BMI; see Exercise 2.30). Write a program that prompts for the person's information, instantiates an object of class `HealthProfile` for that person and prints the information from that object—including the person's first name, last name, gender, date of birth, height and weight—then calculates and prints the person's age in years, BMI, maximum heart rate and target-heart-rate range. It should also display the BMI values chart from Exercise 2.30.

**3.17** *(Automating Electric Energy Purchases)* The Tanzania Electric Supply Company Limited (Tanesco) is a parastatal organization established in 1964 to generate, distribute and market electricity for domestic and industrial use. In recent years, Tanesco has been phasing out its old analog meters and introducing new digital meters which enable customers to buy electricity according to their energy needs. This approach has reduced Tanesco's costs considerably.

Currently, customers have several options to buy electricity. They can go to a vendor and pay in cash, they can pay through their mobiles using either their bank account or using a service called mobile money. When paying for electricity, several deductions are made on the paid amount. These include an 18% VAT, a fixed monthly service charge (5000 Tanzanian Shillings), a 3% addition for the Rural Electricity Agency (REA), and a 1% addition for the Electricity and Water Utility Regulatory Authority (EWURA). The remaining amount after these deductions is used to purchase energy (in kilowatt hours of electricity).

Write a class called `ElectricBill` that will automate the process of buying electricity. The class should have one data member called `amountPaid` (type `double`) and a constructor which receives one parameter for initializing `amountPaid`. Provide *get* and *set* methods for `amountPaid`. If the value of a parameter is less than zero, the value of `amountPaid` should be set to `0`. Also, provide the following methods returning a `double`: `getVAT`, `getEWURA`, `getREA` and `getUnits`, where the first three methods return the amount of the corresponding deduction on the paid amount, and the last method returns the number of kilowatt hours a customer gets, calculated by dividing by the price of one kilowatt hour, the amount remaining after all the deductions. Assume one kilowatt hour is sold at 295 Tanzanian Shillings. Write a driver program to demonstrate the capabilities of class `ElectriBill`.