# Introduction to Classes, Objects, Member Functions and Strings

Chapter 3 of C++ How to Program, 10/e

# OBJECTIVES

In this chapter you'll:

- Begin programming with the object-oriented concepts introduced in Section .
- 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.

# 3.1 Introduction

- Simple bank-account 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.

# 3.1 Introduction (cont.)

- 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.

# 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` (Fig. 3.1) is referred to as a driver program.

```
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 }
```

**Fig. 3.1** | Creating and manipulating an `Account` object. (Part 1 of 2.)

```
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. (Part 2 of 2.)

# 3.2.1 Instantiating an Object

- Typically, you cannot call a member function of a class until you create an object of that class.
- Line 10

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

  creates `myAccount` object of class `Account`.
- The variable's type is `Account` (Fig. 3.2).

# 3.2.2 Headers and Source-Code Files

- The compiler knows what `int` is—it's a fundamental type that's "built into" C++.
- 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.
- 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.

# 3.2.2 Headers and Source-Code Files (cont.)

- 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:

```
#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.

# 3.2.2 Headers and Source-Code Files (cont.)

- 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, 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.
- Can get `myAccount`'s 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.

# 3.2.3 Calling Class Account's getName Member Function (cont.)

- From main's view, when the getName member function is called:
  - The program transfers execution from the call to member function getName.
    - Because getName was called via the myAccount object, getName "knows" which object's data to manipulate.
  - Next, member function getName performs its task—that is, it returns (i.e., gives back) myAccount's name to where the function was called.
    - The main function does not know the details of how getName performs its task.
  - The cout object displays the name returned by member function getName, then the program continues executing with the next statement.

# 3.2.4 Inputting a `string` with `getline`

- `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>`.
- The class name `string`, like the name `cout`, belongs to namespace `std`.

# 3.2.4 Inputting a `string` with `getline` (cont.)

- Sometimes functions are not members of a class.
- Such functions are called global functions.
- Standard Library global function `getline` reads a line of text.
- Like class `string`, function `getline` requires the `<string>` header and belongs to namespace `std`.

# 3.2.4 Inputting a `string` with `getline` (cont.)

- Consider why we cannot simply obtain a full name with

```
cin >> theName;
```

- We entered the name "`Jane Green`," which contains multiple words separated by a space.
- When reading a `string`, `cin` stops at the first white-space character (such as a space, tab or newline).
  - 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.

# 3.2.4 Inputting a `string` with `getline` (cont.)

- 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 its second argument.

# 3.2.5 Calling Class Account's setName Member Function

- Line 19

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

  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 is the argument that's passed to setName, which stores theName's value in the object myAccount.

# 3.2.5 Calling Class `Account`'s `setName` Member Function (cont.)

- From `main`'s view, when `setName` is called:
  - The program transfers execution from the call in `main` to the `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.
  - Next, member function `setName` stores the argument's value in the `myAccount` object.
  - When `setName` completes execution, program execution returns to where `setName` was called, then continues with the next statement.

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

- This section presents class `Account`'s details and 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.
- 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
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.

# 3.3.2 Keyword `class` and the Class Body

- The class definition begins with

```
class Account {
```

- Keyword `class` is followed immediately by the class's name.
- Every class's body is enclosed in an opening left brace and a closing right brace.
- The class definition terminates with a required semicolon.
- For reusability, place each class definition in a separate header with the `.h` filename extension.

## Common Programming Error 3.1

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

# 3.3.2 Keyword `class` and the Class Body (cont.)

- 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`

- 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 that manipulate the data members belonging to particular objects of the class.

# 3.3.3 Data Member `name` of Type `string` (cont.)

- Data members are declared inside a class definition but outside the bodies of the class's member functions.
- The following declares data member `name` of type `string`.

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

- A data member can be manipulated by each of the class's member functions.
- The default value for a `string` is the empty `string` (i.e., `""`).

## 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.*

# 3.3.3 Data Member `name` of Type `string` (cont.)

- 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

- The first line of each function definition is 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` indicates that a function does not return (i.e., give back) any information to its calling function.

# 3.3.4 `setName` Member Function (cont.)

- Car analogy mentioned that pressing a car's gas pedal sends a message to the car to perform a task—make the car go faster.
  - 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.
- A member function can require one or more parameters that represent the data it needs to perform its task.
- When the following statement 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

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

# 3.3.4 `setName` Member Function (cont.)

- Parameters are declared in a parameter list located in the required parentheses following the member function's name.
- Each parameter must specify a type followed by a parameter name.
- Parameters are separated by a comma, as in

```
(type1 name1, type2 name2, …)
```

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

# 3.3.4 `setName` Member Function (cont.)

- Every member 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 member function's task(s).
- When program execution reaches the member function's closing brace, the function returns to its caller.

# 3.3.4 setName Member Function (cont.)

- 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.

# 3.3.5 getName Member Function

- 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.
- The `return` statement passes a value back to the caller, which then can use the returned value.
- We declared member function `getName` as `const` (after the parameter list) because the function does not, and should not, modify the `Account` object on which it's called

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

## 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`

- `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.
  - 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.
- Data members or member functions listed after the `public` 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.

# 3.3.6 Access Specifiers `private` and `public` (cont.)

- 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.
- 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

- UML class diagrams 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`.

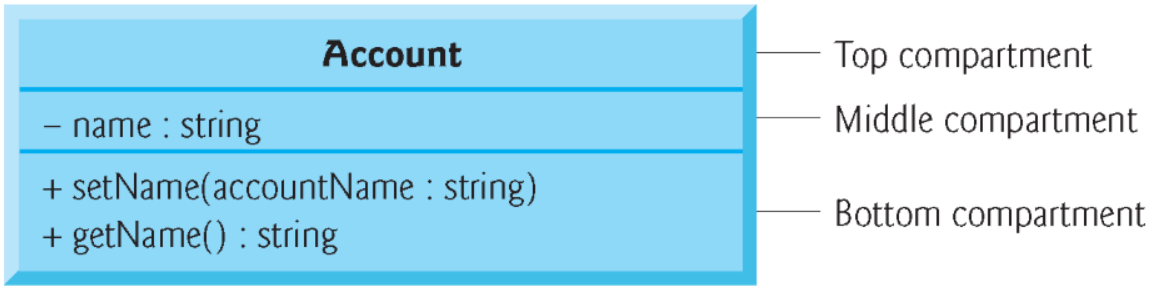


**Fig. 3.3** | UML class diagram for class `Account` of Fig. 3.2.

# 3.3.7 `Account` UML Class Diagram (cont.)

- 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 attributes, which correspond to the data members of the same name in C++.
- The UML class diagram lists a minus sign (–) access modifier before the attribute name for `private` attributes (or other `private` members).
- Following the attribute name are a colon and the attribute type.

# 3.3.7 `Account` UML Class Diagram (cont.)

- The bottom compartment contains the class's operations, 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.
- A plus sign (+) indicates `public` in the UML.
- The UML indicates the return type of an operation by placing a colon and the return type after the parentheses following the operation name.

# 3.3.7 Account UML Class Diagram (cont.)

- 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.

# 3.4 Account Class: Initializing Objects with Constructors

- Each class can define a constructor that specifies custom initialization for objects of that class.
  - Special member function that must have the same name as the class.
- C++ requires a constructor call when each object is created—ideal point to initialize an object's data members.
- A constructor can have parameters—the corresponding argument values help initialize the object's data members.

# 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.
- `Account`'s constructor definition

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

- Normally, constructors are `public`.
- A constructor's parameter list specifies pieces of data required to initialize an object.

```
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`.

# 3.4.1 Defining an `Account` Constructor for Custom Object Initialization (cont.)

- A member-initializer list initializes data members (typically with argument values):

```
: name{accountName}
```

- Member initializers appear between a constructor's parameter list and the left brace that begins the constructor's body.
- 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.
- 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.*

# 3.4.1 Defining an `Account` Constructor for Custom Object Initialization (cont.)

- We declared this constructor `explicit`, because it takes a single parameter—important for subtle reasons that you'll learn in later chapters.
  - For now, declare all single-parameter constructors `explicit`.
- Constructors cannot specify return types
  - not even `void`.
- Constructors cannot be declared `const`
  - Initializing an object modifies it.

# 3.4.1 Defining an `Account` Constructor for Custom Object Initialization (cont.)

- Fig. 3.4: Constructor and `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.

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

- 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.
- Lines 15–16 use each object's `getName` member function to obtain the names and show that they were initialized when the objects were created.
- The output shows different names, confirming that each `Account` maintains its own `name`.

```
1  // Fig. 3.5: AccountTest.cpp
2  // Using the Account constructor to initialize the name data
3  // member at the time each Account object is created.
4  #include <iostream>
5  #include "Account.h"
6
7  using namespace std;
8
9  int main() {
10    // create two Account objects
11    Account account1{"Jane Green"};
12    Account account2{"John Blue"};
13
14    // display initial value of name for each Account
15    cout << "account1 name is: " << account1.getName() << endl;
16    cout << "account2 name is: " << account2.getName() << endl;
17 }
```

```
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.

# 3.4.2 Initializing `Account` Objects When They're Created (cont.)

- 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.
  - 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.

# 3.4.2 Initializing `Account` Objects When They're Created (cont.)

- 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.
  - 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.

## 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, 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.

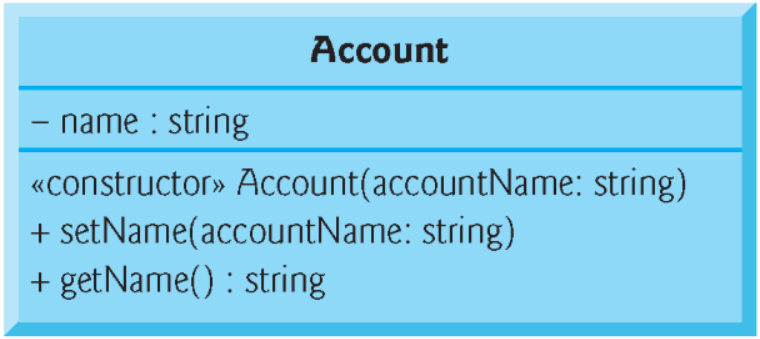


**Fig. 3.6** | UML class diagram for the `Account` class of Fig. 3.4.

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

- Set and get member functions can validate attempts to modify `private` data and control how that data is presented to the caller, respectively—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.

# 3.5 Software Engineering with *Set* and *Get* Member Functions (cont.)

- *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.

# 3.5 Software Engineering with *Set* and *Get* Member Functions (cont.)

- A get function can present the data in a different form, while the actual data representation remains hidden from the user.
  - 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.

# 3.5 Software Engineering with *Set* and *Get* Member Functions (cont.)

- 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.

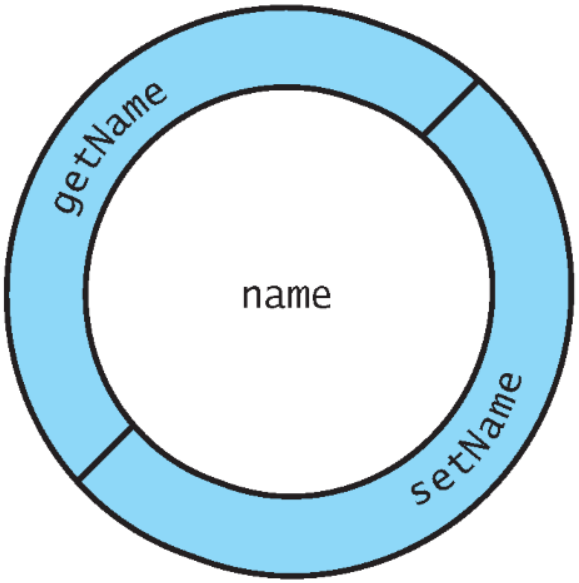


**Fig. 3.7** | Conceptual view of an `Account` object with its encapsulated `private` data member `name` and protective layer of `public` member functions.

**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`.
- 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 declares a data member `balance` of type `int` and initializes its value to `0`

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

- This is known as an in-class initializer and was introduced in C++11.

```
1  // Fig. 3.8: Account.h
2  // Account class with name and balance data members, and a
3  // constructor and deposit function that each perform validation.
4  #include <string>
5
6  class Account {
7  public:
8     // Account constructor with two parameters
9     Account(std::string accountName, int initialBalan(
10       : name{accountName} { // assign accountName to data member name
11
12       // validate that the initialBalance is greater than 0; if not,
13       // data member balance keeps its default initial value of 0
14       if (initialBalance > 0) { // if the initialBalance is valid
15          balance = initialBalance; // assign it to data member balan
16       }
17    }
18
```

**Fig. 3.8** | `Account` class with `name` and `balance` data members, and a constructor and `deposit` function that each perform validation. (Part 1 of 3.)

```
19      // function that deposits (adds) only a valid amount to the balance
20      void deposit(int depositAmount) {
21         if (depositAmount > 0) { // if the depositAmount is valid
22            balance = balance + depositAmount; // add it to the balance
23         }
24      }
25
26      // function returns the account balance
27      int getBalance() const {
28         return balance;
29      }
30
31      // function that sets the name
32      void setName(std::string accountName) {
33         name = accountName;
34      }
35
36      // function that returns the name
37      std::string getName() const {
38         return name;
39      }
```

**Fig. 3.8** | `Account` class with `name` and `balance` data members, and a constructor and `deposit` function that each perform validation. (Part 2 of 3.)

```
40   private:
41      std::string name; // account name data member
42      int balance{0}; // data member with default initial value
43   }; // end class Account
```

**Fig. 3.8** | `Account` class with `name` and `balance` data members, and a constructor and `deposit` function that each perform validation. (Part 3 of 3.)

# 3.6.1 Data Member `balance` (cont.)

- 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

- 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.

# 3.6.2 Two-Parameter Constructor with Validation (cont.)

- Lines 14–16 of Fig. 3.8 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 (initialBalance > 0) { // if the initialBalance is valid
   balance = initialBalance; // assign it 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 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.

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

# 3.6.3 `deposit` Member Function with Validation (cont.)

- Line 22 first adds the current `balance` and `depositAmount`, forming a temporary sum which is then assigned to `balance`, replacing its prior value
- It's important to understand that the calculation `balance + depositAmount` on the right side of the assignment operator does not modify the `balance`—that's why the assignment is necessary.

# 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.
- `getBalance` is declared `const`, because 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 `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.
- 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 can create code maintenance prob.lems
  - 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 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`.

```
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
```

**Fig. 3.9** | Displaying and updating `Account` balances. (Part 1 of 3.)

```
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 2 of 3.)

```
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 3 of 3.)

## 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.*

## 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.
- Second compartment contains the `private` attributes
  - `name` of type `string`
  - `balance` of type `int`.
- Constructor is modeled in the third compartment
  - With parameters `accountName` of type `string` and `initialBalance` of type `int`
- The class's `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`
  - operation `getName` with a return type of `string`.

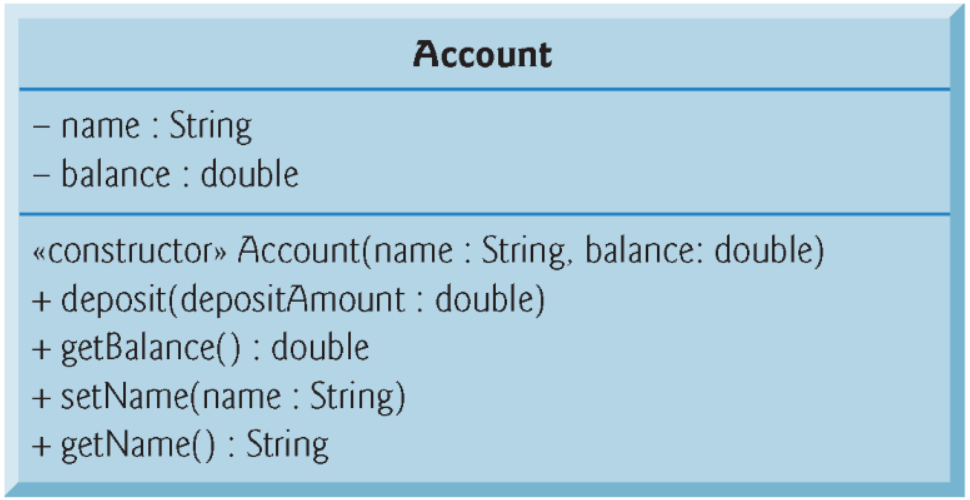


**Fig. 3.10** | UML class diagram for the `Account` class of Fig. 3.8.