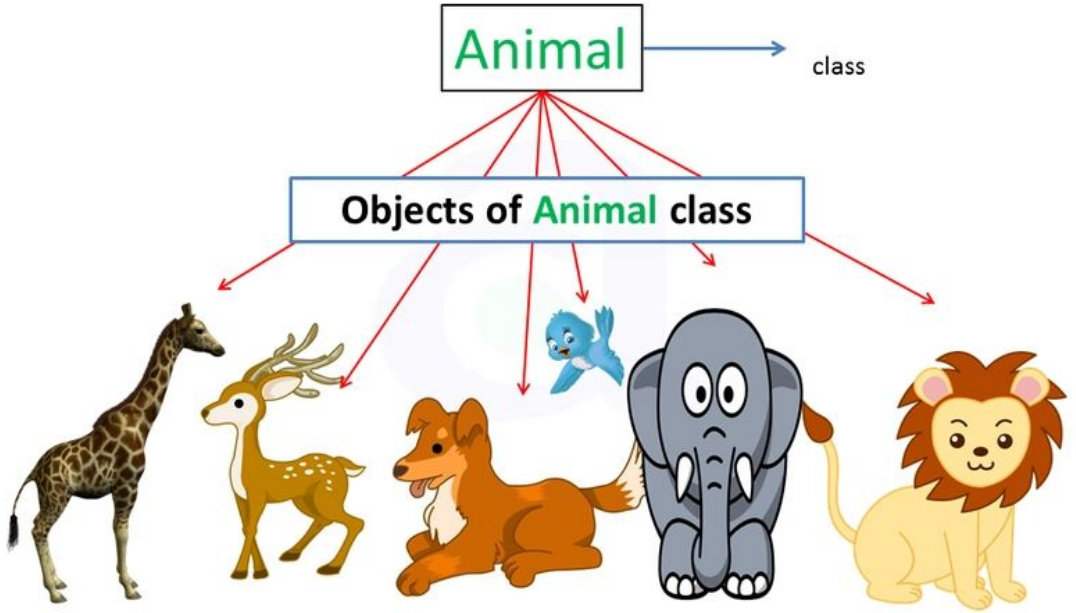
Introduction to Classes, Objects, Member Functions, and Strings

Object-Oriented Programming

- Object Oriented Programming (OOP) is a programming paradigm centered around the concept of ‘Objects’.
- In OOP, everything in programming is structured as objects and classes.
- These objects and classes serve as the foundations or grammar of the language, enabling communication with other devices to achieve specific tasks or goals.



Advantages of OOPS

- Modularity through classes
- Code Reusability
- Encapsulation
- Inheritance
- Abstraction
- Polymorphism
- Data maintenance and Security
- Design advantages

Modularity through Classes

- Break a big program into smaller, manageable pieces called *classes*.
- Example:

```
class Student {
public:
   string name;
   int rollNo;
   void display() {
      cout << "Name: " << name << ", Roll No: " << rollNo << endl;
   }
};
```

- Here, the student class is a module that can be worked on independently without affecting other parts of the program.

Code Reusability

- Write once, use multiple times.
- Example:

```
class Calculator {
public:
    int add(int a, int b) { return a + b; }
    int sub(int a, int b) { return a - b; }
};
```

- The calculator class can be reused in many programs (marks calculation, billing system, etc.) without rewriting code.

Encapsulation

- In general, encapsulation is a process of wrapping similar code in one place.
- In C++, we can bundle data members and functions that operate together inside a single class.

```
class Rectangle {
    public:
        int length;
        int breadth;

        int getArea() {
            return length * breadth;
        }
};
```

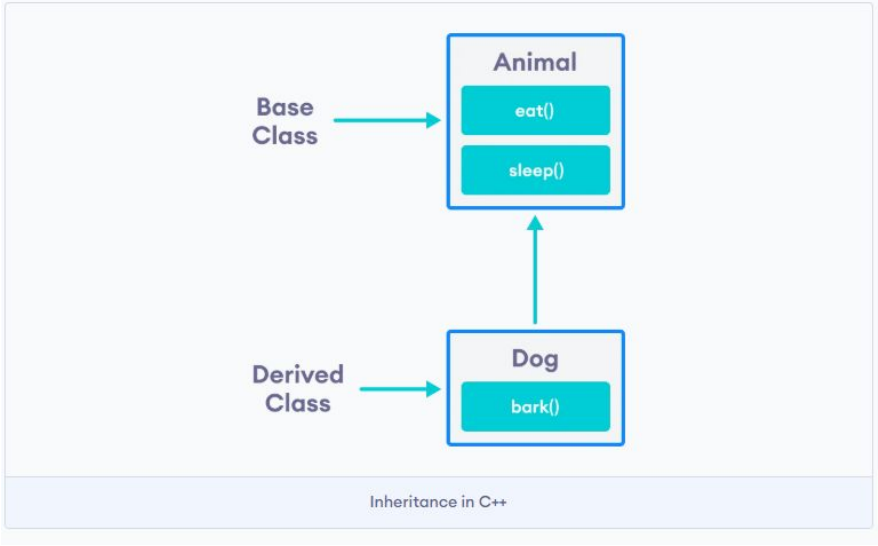
- In the above program, the function getArea() calculates the area of the rectangle. To calculate the area, it needs length and breadth.
- Hence, the data members (length and breadth) and the function getArea() are kept together in the Rectangle class.

Inheritance

- Inheritance allows us to create a new class (derived class) from an existing class (base class).
- **The derived class inherits the features from the base class** and can have additional features of its own.

```
class Animal {
    // eat() function
    // sleep() function
};

class Dog : public Animal {
    // bark() function
};
```



- Here, the Dog class is derived from the Animal class. Since Dog is derived from Animal, members of Animal are accessible to Dog.
- Notice the use of the keyword public while inheriting Dog from Animal. We can also use the keywords private and protected instead of public.

Inheritance

is-a relationship

- Inheritance is an **is-a relationship**. We use inheritance only if an **is-a relationship** is present between the two classes.
- Here are some examples:
  - A car is a vehicle.
  - Orange is a fruit.
  - A surgeon is a doctor.
  - A dog is an animal.

Abstraction

- Abstraction in C++ involves hiding the complex implementation details and showing only the essential features of an object.
- Example:

```
class Printer {
public:
    void printDocument(string text) {
        // Complex logic hidden from user
        cout << "Printing: " << text << endl;
    }
};
```

- When students use printDocument(), they don’t need to know how printing works internally.

Polymorphism

- Polymorphism simply means more than one form. That is, the same entity (function or operator) behaves differently in different scenarios.
- For example, the + operator in C++ is used to perform two specific functions. When it is used with numbers (integers and floating-point numbers), it performs addition.

```
int a = 5;
int b = 6;
int sum = a + b;    // sum = 11
```

- And when we use the + operator with strings, it performs string concatenation.

```
string firstName = "abc ";
string lastName = "xyz";

// name = "abc xyz"
string name = firstName + lastName;
```

- We can implement polymorphism in C++ using the following ways:
  - Function Overloading
  - Operator Overloading
  - Function Overriding
  - Virtual Functions

Polymorphism

- Polymorphism allows us to create consistent code. For example,
- Suppose we need to calculate the area of a circle and a square.
- To do so, we can create a Shape class and derive two classes Circle and Square from it.
- In this case, it makes sense to create a function having the same name calculateArea() in both the derived classes rather than creating functions with different names, thus making our code more consistent.

Data maintenance

- Easier to fix or improve code without affecting other parts.

```
#include <iostream>
using namespace std;
class Temperature {
private:
   double celsius;
public:
   void setCelsius(double c) { celsius = c; }
   double getCelsius() { return celsius; }
   // This function can be improved later without changing rest of the program
   double toFahrenheit() { return (celsius * 9.0/5.0) + 32; }
};
int main() {
   Temperature t;
   t.setCelsius(25);
   cout << "Fahrenheit: " << t.toFahrenheit();
}
```

- If the formula changes or you want to log conversions, you only modify toFahrenheit () – no need to rewrite the whole program.

Data Security

- Prevents unauthorized access to sensitive data.

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

class BankAccount {
private:
    string password;
    double balance;

public:
    BankAccount(string pass, double bal) {
        password = pass;
        balance = bal;
    }

    double getBalance(string pass) {
        if (pass == password)
            return balance;
        else {
            cout << "Access Denied!\n";
            return -1;
        }
    }
};

int main() {
    BankAccount acc("abcd1234", 5000);

    cout << acc.getBalance("abcd1234") << endl; // Correct password
    cout << acc.getBalance("wrongpass") << endl; // Wrong password
}
```

- Only people with the correct password can view the balance.

Design advantages

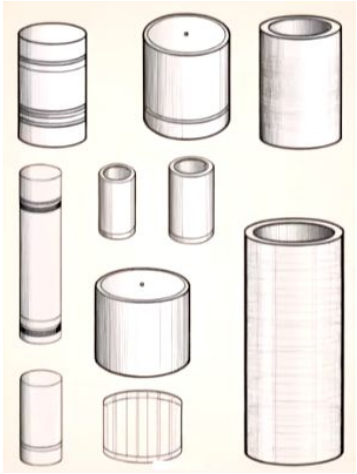
- Easier to understand, maintain, and extend programs.
- Examples:
  - If you design a Library Management System with classes like Book, Member, Librarian, you can add new features (like online renewals) without rewriting the entire code.
  - If you design a College Attendance System with classes like Student, Course, and AttendanceRecord, you can later add new features (like automatic attendance using face recognition) without changing the existing classes — just create a new FaceRecognitionAttendance class that works with them.

Struct vs. Class

- In C++, no difference between struct and class (except default public vs. private)
- In C++, a struct can have:
  - member variables
  - methods
  - public, private, and protected
  - virtual functions
  - etc.
- Rule of thumb:
  - Use a struct when member variables are public (just a container)
  - Use class otherwise

Classes and Objects in C++

- In the object-oriented programming approach, programs are designed using objects and classes.
  - An object is an entity that encapsulates both state and behaviour, where state refers to data and behavior refers to the functionality.
  - Objects are instantiated at runtime.



Class: Cylinder

- State refers to member data.
- Behavior refers to member functions.

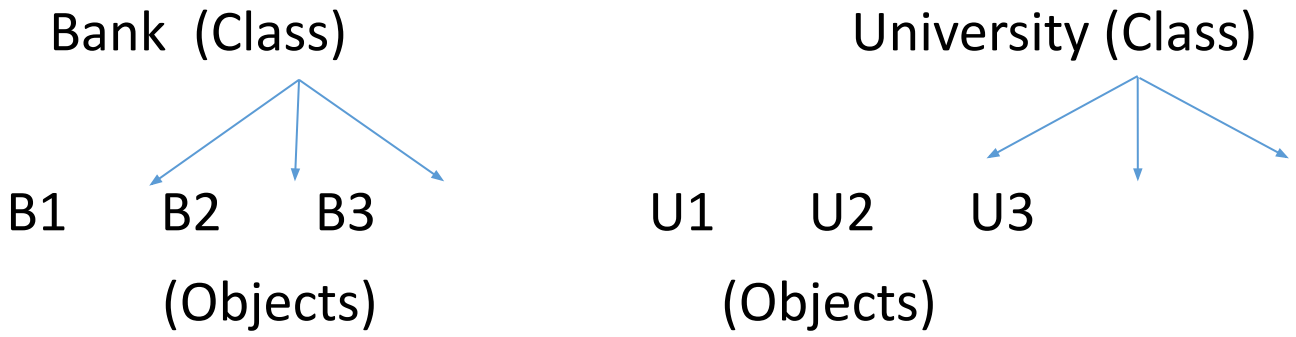
Objects in C++

- An object is an instance of a class, through which all members of a class can be accessed.

Time T1;

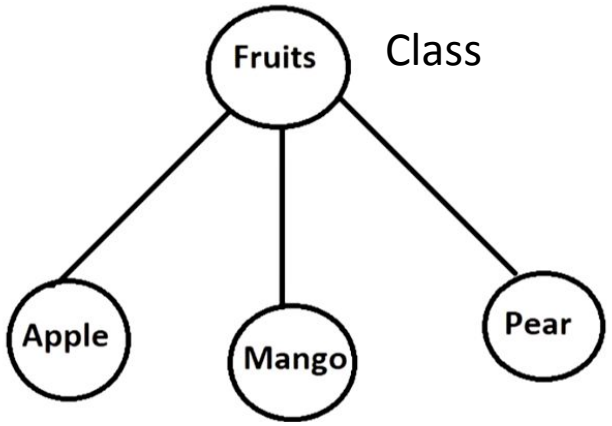
Class Object

- Objects can represent real-world entities such as a chair, bike, marker, pen, table, or car.
- They can be either physical (tangible) or logical (intangible).
- For instance, a banking system is an example of an intangible object.



Objects in C++

- An object has three characteristics:
  - State: Represents data (value) of an object.
  - Behavior: Represents the behavior (functionality) of an object, such as deposit, withdraw, etc.
  - Identity: Object identity is typically implemented via a unique ID.
- For example, a Pen is an object. Its name is Parker, color is Golden etc. known as its state. It is used to write, so writing is its behavior.
- An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance (result) of a class.
- A class may contain several objects.



Example:

```
Class A{
        int a;
        double b[10];
        char c;
        int fun1();
        …
        …
}
```

Classes in C++

- A class is a group of similar objects.
- It is a template from which objects are created.
- It can have fields, functions, constructors, etc.
- A class in C++ can contain:
  - Member data
  - Member functions
  - Constructor
  - Block
  - Class
- C++ class

```
Class <class_name>{
        member data;  // field
        member function();
};
```

Object 1 Object 2 Object 3

Classes naming Conventions

- In **OOP**, class naming conventions are mostly about making your code **readable, consistent, and professional**.

1. **Use PascalCase(UpperCamelCase)**
   - Each word starts with a capital letter, no underscores.
   - Examples:
     - StudentRecord ✔
     - BankAccount ✔
     - librarymanagementsystem ❌(hard to read)
     - bank_account ❌ (snake_case is usually for variables in some languages)
2. **Class Names Should be Nouns**
   - Classes represent **entities or concepts**, so names should be nouns, not verbs.
   - Examples
     - Car, Teacher, InvoiceGenerator ✔
     - Calculate, Drive, Print ❌ (verbs are for methods)

Classes naming Conventions

3. **Be Descriptive, Not Abbreviated**
   - Names should clearly convey the purpose of the class.
   - **Examples**
     - CustomerDatabase ✔
     - CustDB ❌ (unclear unless you are the original author)
4. **Avoid Special Characters**
   - Only use letters and digits (no spaces, no -, no _, for class names in OOP convention)
5. **Use Singular Form**
   - Unless the class represents a collection.
   - **Examples**
     - Book ✔
     - Books ❌ (unless it's a Books collection class)
6. **Match Class Name with File Name** (especially in Java)
   - In Java, a public class StudentRecord must be in a file named StudentRecord.java
   - C++ does not enforce this, but it's a good practice.

Classes naming Conventions

- Examples Following Conventions

```
class StudentRecord { };
class BankAccount { };
class LibraryManagementSystem { };
```

- Bad Examples

```
class stu_rec { };          // Not descriptive, wrong case
class CALCULATOR { };       // All caps = constants
class dataProcess { };      // Verb instead of noun
```

Example of a Class

```
// Creating class is like planning
// You have to execute a plan with functionalities using an object, which is like building a home.
class Home{
        // you can plan functionalities here
        void designHall()
        {
                // code …..
        }
        void designKitchen()
        {
                // code …..
        }
};
```

Introduction

- 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.
  - You'll add the withdraw member function.

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.

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.)

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).

Headers and Source-Code Files

- When we declare variables of type int, the compiler knows what int is—it's a fundamental type that's "built into" C++.
- 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.
- 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.

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 the header is in the same folder as Fig. 3.1, rather than with the C++ Standard Library headers.

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.
- You include headers into source-code files, though you also may include them in other headers.

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.

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

- From the main's view, when the getName member function is called:
  - The program transfers execution from the call (line 13 in main) to the 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 line 13, 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 (at line 16 in main).
  - In this case, line 13 does not display a name, because we have not yet stored a name in the myAccount object.

Inputting a `string` with `getline`

- Line 17
  - String theName;
- 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 the 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

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

- 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 the namespace std.

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

- 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 the namespace std.

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

- Consider why we cannot simply write,

  cin >> theName;

  to obtain the account name.
- In sample program execution, 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.

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

- In this example, we would like the user to type the complete name (including the space) and press *Enter* to submit it to the program.
- Then, we would 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.

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

- From main's view, when setName is called:
  - The program transfers execution from the call (line 19) 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 (line 19), then continues with the next statement (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 the 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.

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

- This section presents the class Account's details and a UML diagram that summarizes the class Account's attributes and operations in a concise graphical representation.

`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.

Keyword `class` and the Class Body

- The class definition begins in line 6:

  class Account {

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

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.

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.
- 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 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 (Line 18).

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

- 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., "").
- Good programming practice
  - By convention, place a class's data members last in the class's body. You can list the class's data members any where in the class outside its member-function definitions, but scattering the data members can lead to hard to read code.

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).

`setName` Member Function

- The first line of each function definition (Line 9) 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 (line 9) indicates that when setName completes its task, it does not return (i.e., give back) any information to its calling function – line 19 of the main function.

`setName` Member Function (cont.)
setName Parameter

- 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.
- 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 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 (line 9).

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

`setName` Member Function (cont.) 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 (cont.)** setName Member Function Body

- 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).
- In our example, the member function body contains a single statement (line 10)

  name = accountName; // stores the account name

- That assigns the accountName parameter's value (a string) to the class's data member, thus storing the account name in the object for which setName was called-my Account in this example's main program.
- When program execution reaches the member function's closing brace (line 11), the function returns to its caller.

`setName` Member Function (cont.)

- Parameters are local variables
  - 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 are also local variables of that function.

`setName` Member Function (cont.)

- 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.
  - 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 the 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.

`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 a data member back to the caller, which then can use the returned value.

getName Member Function

- 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 (after the parameter list) 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.
- 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.

`getName` Member Function

- 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 (after the parameter list) 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

Access Specifiers `private` and `public`

- The keyword 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 (such as main), and by member functions of other classes.

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

- 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 can prefer to list public only once, grouping everything that's public, and we can 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
  - 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
  - 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.

Account UML Class Diagram

- UML (Unified Modeling Language) class diagrams are often used 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.

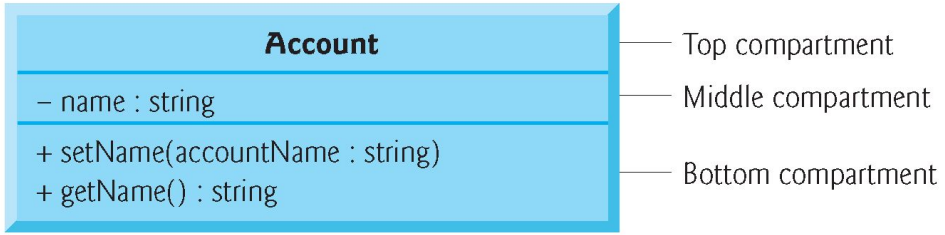


Fig. 3.3 | UML class diagram for class Account of Fig. 3.2.

`Account` UML Class Diagram (cont.)

- In the UML, each class is modeled in a class diagram as a rectangle with three compartments.
- Top compartment
  - The top compartment contains the class name centered horizontally in boldface type.
- Middle compartment
  - 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 is a colon and the attribute type, in this case, string.

`Account` UML Class Diagram (cont.)

- 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.

`Account` UML Class Diagram (cont.)

- 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.

`Account` UML Class Diagram (cont.)

- Parameters
  - The UML models a parameter by listing the parameter name, followed by a colon and the parameter type in 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