Chapter 6
Defining Classes

Overview

• Introduction
• Structures (6.1)
• Classes (6.2)
• Abstract Data Types (6.3)

What Is a Class?

• A class is a data type whose variables are objects
• Some pre-defined classes you have used are
  • int
  • char
  • ifstream
• You can define your own classes as well

Copyright © 2005 Pearson Addison-Wesley. All rights reserved.

Copyright © 2005 Pearson Addison-Wesley. All rights reserved.
Using the Structure

- Structure definition is generally placed outside any function definition
  - This makes the structure type available to all code that follows the structure definition
- To declare two variables of type CDAccount:

  ```
  CDAccount my_account, your_account;
  ```
  - My_account and your_account contain distinct member variables balance, interest_rate, and term

Using Member Variables

- Member variables can be used just as any other variable of the same type
  - my_account.balance = 1000;
    your_account.balance = 2500;
  - Notice that my_account.balance and your_account.balance are different variables!
  - my_account.balance = my_account.balance + interest;

The Structure Value

- The Structure Value
  - Consists of the values of the member variables
- The value of an object of type CDAccount
  - Consists of the values of the member variables
    ```
    balance
    interest_rate
    term
    ```

Duplicate Names

- Member variable names duplicated between structure types are not a problem.
  - struct FertilizerStock
    ```
    {
      double quantity;
      double nitrogen_content;
    }
    ```
  - struct CropYield
    ```
    {
      int quantity;
      double size;
    }
    ```
  - FertilizerStock super_grow;
  - CropYield apples;
  - super_grow.quantity and apples.quantity are different variables stored in different locations

Specifying Member Variables

- Member variables are specific to the structure variable in which they are declared
  - Syntax to specify a member variable: Structure_Variable_Name.Member_Variable_Name
  - Given the declaration:
    ```
    CDAccount my_account, your_account;
    ```
  - Use the dot operator to specify a member variable
    ```
    my_account.balance
    my_account.interest_rate
    my_account.term
    ```

Structures as Arguments

- Structures can be arguments in function calls
  - The formal parameter can be call-by-value
  - The formal parameter can be call-by-reference
- Example:
  ```
  void get_data(CDAccount& the_account);
  ```
  - Uses the structure type CDAccount we saw earlier as the type for a call-by-reference parameter
Structures as Return Types

- Structures can be the type of a value returned by a function.
- Example:
  ```c
  struct CDAccount shrink_wrap(double the_balance, double the_rate, int the_term)
  {
      CDAccount temp;
      temp.balance = the_balance;
      temp.interest_rate = the_rate;
      temp.term = the_term;
      return temp;
  }
  ```

Hierarchical Structures

- Structures can contain member variables that are also structures.
- ```
  struct Date
  {
      int month;
      int day;
      int year;
  };

  struct PersonInfo
  {
      double height;
      int weight;
      Date birthday;
  };

  struct PersonInfo contains a Date structure
  ```

Using Function `shrink_wrap`

- `shrink_wrap` builds a complete structure value in `temp`, which is returned by the function.
- We can use `shrink_wrap` to give a variable of type `CDAccount` a value in this way:
  ```c
  CDAccount new_account;
  new_account = shrink_wrap(1000.00, 5.1, 11);
  ```

Using `PersonInfo`

- A variable of type `PersonInfo` is declared by:
  ```
  PersonInfo person1;
  ```
- To display the birth year of `person1`, first access the birthday member of `person1`:
  ```
  cout <<  person1.birthday.
  ```
- But we want the year, so we now specify the year member of the birthday member:
  ```
  cout << person1.birthday.year;
  ```

Assignment and Structures

- The assignment operator can be used to assign values to structure types.
- Using the `CDAccount` structure again:
  ```c
  CDAccount my_account, your_account;
  my_account.balance = 1000.00;
  my_account.interest_rate = 5.1;
  my_account.term = 12;
  your_account = my_account;
  ```

  Assigns all member variables in `your_account` the corresponding values in `my_account`.

Initializing Classes

- A structure can be initialized when declared.
- Example:
  ```c
  struct Date
  {
      int month;
      int day;
      int year;
  };

  Date due_date = {12, 31, 2004};
  ```
Section 6.1 Conclusion

• Can you

  • Write a definition for a structure type for records consisting of a person’s wage rate, accrued vacation (in whole days), and status (hourly or salaried). Represent the status as one of the two character values ‘H’ and ‘S’. Call the type EmployeeRecord.

Class DayOfYear Definition

  • class DayOfYear
    {
    public:
    void output( );
    int month;
    int day;
    
    \[ \text{Member Function Declaration} \]

6.2 Classes

• A class is a data type whose variables are objects
  • The definition of a class includes
    • Description of the kinds of values of the member variables
    • Description of the member functions
  • A class description is somewhat like a structure definition plus the member variables

Defining a Member Function

• Member functions are declared in the class declaration
• Member function definitions identify the class in which the function is a member
  • void DayOfYear::output()
    {
    cout << "month = " << month
    << ",  day = " << day
    << endl;
    
    \[ \text{Member Function Definition} \]

A Class Example

• To create a new type named DayOfYear as a class definition
  • Decide on the values to represent
  • This example’s values are dates such as July 4 using an integer for the number of the month
    • Member variable month is an int (Jan = 1, Feb = 2, etc.)
    • Member variable day is an int
  • Decide on the member functions needed
  • We use just one member function named output

Member Function Definition

• Member function definition syntax:
  Returned_Type
  Class_Name::Function_Name( Parameter_List )
  {
  Function Body Statements
  
  \[ \text{Member Function Definition} \]
The ‘::’ Operator

• ‘::’ is the scope resolution operator
  • Tells the class a member function is a member of
  • void DayOfYear::output() indicates that function output is a member of the DayOfYear class
  • The class name that precedes ‘::’ is a type qualifier

‘::’ and ‘.’

• ‘::’ used with classes to identify a member
  void DayOfYear::output() {
    // function body
  }
  • ‘.’ used with variables to identify a member
    DayOfYear birthday;
    birthday.output();

Problems With DayOfYear

• Changing how the month is stored in the class DayOfYear requires changes to the program
  • If we decide to store the month as three characters (JAN, FEB, etc.) instead of an int
    • cin >> today.month will no longer work because we now have three character variables to read
    • if(today.month == birthday.month) will no longer work to compare months
  • The member function "output" no longer works

Encapsulation

• Encapsulation is
  • Combining a number of items, such as variables and functions, into a single package such as an object of a class

Calling Member Functions

• Calling the DayOfYear member function output is done in this way:
  DayOfYear today, birthday;
  today.output();
  birthday.output();
  • Note that today and birthday have their own versions of the month and day variables for use by the output function

Ideal Class Definitions

• Changing the implementation of DayOfYear requires changes to the program that uses DayOfYear
  • An ideal class definition of DayOfYear could be changed without requiring changes to the program that uses DayOfYear
Fixing DayOfYear

- To fix DayOfYear
  - We need to add member functions to use when changing or accessing the member variables
  - If the program never directly references the member variables, changing how the variables are stored will not require changing the program
  - We need to be sure that the program does not ever directly reference the member variables

Public or Private Members

- The keyword private identifies the members of a class that can be accessed only by member functions of the class
  - Members that follow the keyword private are private members of the class
  - The keyword public identifies the members of a class that can be accessed from outside the class
  - Members that follow the keyword public are public members of the class

Public Or Private?

- C++ helps us restrict the program from directly referencing member variables
  - private members of a class can only be referenced within the definitions of member functions
  - If the program tries to access a private member, the compiler gives an error message
  - Private members can be variables or functions

A New DayOfYear

- The new DayOfYear class demonstrated in Display 6.4...
  - Uses all private member variables
  - Uses member functions to do all manipulation of the private member variables
  - Member variables and member function definitions can be changed without changes to the program that uses DayOfYear

Private Variables

- Private variables cannot be accessed directly by the program
  - Changing their values requires the use of public member functions of the class
  - To set the private month and day variables in a new DayOfYear class use a member function such as
    ```cpp
    void DayOfYear::set(int new_month, int new_day)
    {
        month = new_month;
        day = new_day;
    }
    ```

Using Private Variables

- It is normal to make all member variables private
  - Private variables require member functions to perform all changing and retrieving of values
    - Accessor functions allow you to obtain the values of member variables
      - Example: get_day in class DayOfYear
    - Mutator functions allow you to change the values of member variables
      - Example: set in class DayOfYear
General Class Definitions

- The syntax for a class definition is
  ```
  class Class_Name
  {
    public:
      Member_Specification_1
      Member_Specification_2
      Member_Specification_3
    private:
      Member_Specification_n+1
      Member_Specification_n+2
      ...
  };
  ```

Declaring an Object

- Once a class is defined, an object of the class is declared just as variables of any other type
  ```
  class Bicycle
  {
    // class definition lines
  };
  Bicycle my_bike, your_bike;
  ```

The Assignment Operator

- Objects and structures can be assigned values with the assignment operator (=)
  ```
  DayOfYear due_date, tomorrow;
  tomorrow.set(11, 19);
  due_date = tomorrow;
  ```

Program Example: BankAccount Class

- This bank account class allows
  ```
  Withdrawal of money at any time
  All operations normally expected of a bank account
  (implemented with member functions)
  Storing an account balance
  Storing the account’s interest rate
  ```

Calling Public Members

- Recall that if calling a member function from the main function of a program, you must include the object name:
  ```
  account1.update();
  ```

Calling Private Members

- When a member function calls a private member function, an object name is not used
  ```
  fraction(double percent);
  is a private member of the BankAccount class
  fraction is called by member function update
  void BankAccount::update()
  {
    balance = balance + fraction(interest_rate)* balance;
  }
  ```
Constructors

- A constructor can be used to initialize member variables when an object is declared:
  - A constructor is a member function that is usually public.
  - A constructor is automatically called when an object of the class is declared.
  - A constructor’s name must be the name of the class.
  - A constructor cannot return a value.

- No return type, not even void, is used in declaring or defining a constructor.

Constructor Declaration

- A constructor for the BankAccount class could be declared as:
  ```cpp
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    // initialises the balance to dollars.cents
    // initialises the interest rate to rate percent
    ... // The rest of the BankAccount definition
};
```

Calling A Constructor (1)

- A constructor is not called like a normal member function:

```cpp
BankAccount account1;
accounts.BankAccount10, 50, 2.0);
```

Constructor Definition

- The constructor for the BankAccount class could be defined as:
  ```cpp
  BankAccount::BankAccount(int dollars, int cents, double rate)
  {
      if ((dollars < 0) || (cents < 0) || (rate < 0))
      {
          cout << "Illegal values for money or rate\n";
          exit(1);
      }
      balance = dollars + 0.01 * cents;
      interest_rate = rate;
  }
  ```
  - Note that the class name and function name are the same.

Calling A Constructor (2)

- A constructor is called in the object declaration:

```cpp
BankAccount account1(10, 50, 2.0);
```
  - Creates a BankAccount object and calls the constructor to initialize the member variables.

Overloading Constructors

- Constructors can be overloaded by defining constructors with different parameter lists:
  - Other possible constructors for the BankAccount class might be:
    ```cpp
    BankAccount (double balance, double interest_rate);
    BankAccount (double balance);
    BankAccount (double interest_rate);
    BankAccount ();
    ```
The Default Constructor

- A default constructor uses no parameters
- A default constructor for the BankAccount class could be declared in this way

```cpp
class BankAccount
{
    public:
        BankAccount( );
        // initializes balance to $0.00
        // initializes rate to 0.0%
        // The rest of the class definition
};
```

Initialization Sections

- An initialization section in a function definition provides an alternative way to initialize member variables
  - BankAccount::BankAccount( ); balance(0),
    interest_rate(0.0);
    { // No code needed in this example
      // The values in parenthesis are the initial values for the member variables listed
  }

Default Constructor Definition

- The default constructor for the BankAccount class could be defined as
  - BankAccount::BankAccount( )
  { balance = 0;
    rate = 0.0;
  }
- It is a good idea to always include a default constructor even if you do not want to initialize variables

Parameters and Initialization

- Member functions with parameters can use initialization sections
  - BankAccount::BankAccount(int dollars, int cents, double rate)
    : balance (dollars + 0.01 * cents),
    interest_rate(rate)
  { if ((dollars < 0) || (cents < 0) || (rate < 0))
      cout << "Illegal values for money or rate\n";
      exit(1);
    }
- Notice that the parameters can be arguments in the initialization

Calling the Default Constructor

- The default constructor is called during declaration of an object
  - An argument list is not used

```cpp
BankAccount account1;
// uses the default BankAccount constructor

BankAccount account1( ); // is not legal
```

Section 6.2 Conclusion

- Can you
  - Describe the difference between a class and a structure?
  - Explain why member variables are usually private?
  - Describe the purpose of a constructor?
  - Use an initialization section in a function definition?
Abstract Data Types

- A data type consists of a collection of values together with a set of basic operations defined on the values.
- A data type is an Abstract Data Type (ADT) if programmers using the type do not have access to the details of how the values and operations are implemented.

Classes To Produce ADTs

- To define a class so it is an ADT
  - Separate the specification of how the type is used by a programmer from the details of how the type is implemented
  - Make all member variables private members
  - Basic operations a programmer needs should be public member functions
  - Fully specify how to use each public function
  - Helper functions should be private members

ADT Implementation

- The ADT implementation tells how the interface is realized in C++
  - The implementation consists of
    - The private members of the class
    - The definitions of public and private member functions
    - The implementation is needed to run a program
    - The implementation is not needed to write the main part of a program or any non-member functions

ADT Interface

- The ADT interface tells how to use the ADT in a program
  - The interface consists of
    - The public member functions
    - The comments that explain how to use the functions
    - The interface should be all that is needed to know how to use the ADT in a program

Program Example

The BankAccount ADT

- In this version of the BankAccount ADT
  - Data is stored as three member variables
    - The dollars part of the account balance
    - The cents part of the account balance
    - The interest rate
  - This version stores the interest rate as a fraction
  - The public portion of the class definition remains unchanged from the version of Display 6.6

ADT Benefits

- Changing an ADT implementation does require changing a program that uses the ADT
- ADT's make it easier to divide work among different programmers
  - One or more can write the ADT
  - One or more can write code that uses the ADT
- Writing and using ADTs breaks the larger programming task into smaller tasks
Interface Preservation

• To preserve the interface of an ADT so that programs using it do not need to be changed
  • Public member declarations cannot be changed
  • Public member definitions can be changed
  • Private member functions can be added, deleted, or changed

Information Hiding

• Information hiding was referred to earlier as writing functions so they can be used like black boxes
• ADT’s implement information hiding because
  • The interface is all that is needed to use the ADT
  • Implementation details of the ADT are not needed to know how to use the ADT
  • Implementation details of the data values are not needed to know how to use the ADT

Section 6.3 Conclusion

• Can you
  • Describe an ADT?
  • Describe how to implement an ADT in C++?
  • Define the interface of an ADT?
  • Define the implementation of an ADT?
Display 6.2

```cpp
// Member Values
int balance;
float interest_rate;
int account_type;
```

Display 6.3

```cpp
// Example code:

int main()
{
    int balance;
    float interest_rate;
    int account_type;

    // Initialize variables
    balance = 0;
    interest_rate = 0.05;
    account_type = 1;

    // Display values
    cout << "Balance: \$" << balance << endl;
    cout << "Interest Rate: \$" << interest_rate << endl;
    cout << "Account Type: \" << account_type << endl;
}
```

Display 6.4

```cpp
// Sample Dialogue

Sample Dialogue:

Enter today's date:
Enter the month as a number: 3
Enter the day of the month: 15
Your birthday is month = 3, day = 15.

Today's date is month = 3, day = 21.
J. S. Bach's birthday is month = 3, day = 21.
Happy Birthday Johann Sebastian!
```
Display 6.6
(3/3)

Class with Constructors (part 3 of 3)

Screen Output
account1 initialized as follows:
Account balance $100.00
Interest rate 2.30%
account2 initialized as follows:
Account balance $0.00
Interest rate 0.00%
account1 reset to the following:
Account balance $999.99
Interest rate 5.50%

Display 6.7
(1/4)

Alternative BankAccount Class Implementation (part 1 of 4)

```cpp
// User faststream
void BankAccount::output(ostream & outs) {
    outs.setf(ios::fixed);
    outs.setf(ios::showpoint);
    outs.precision(2);  
    outs << "Account balance $" << get_balance() << endl;
    outs << "Interest rate “" << get_rate() << “" << endl;
}
```

Display 6.7
(2/4)

Alternative BankAccount Class Implementation (part 2 of 4)

```cpp
// User faststream
void BankAccount::output(ostream & outs) {
    outs.setf(ios::fixed);
    outs.setf(ios::showpoint);
    outs.precision(2);  
    outs << "Account balance $" << get_balance() << endl;
    outs << "Interest rate “" << get_rate() << “" << endl;
}
```

Display 6.7
(3/4)

Alternative BankAccount Class Implementation (part 3 of 4)

```cpp
// User faststream
void BankAccount::output(ostream & outs) {
    outs.setf(ios::fixed);
    outs.setf(ios::showpoint);
    outs.precision(2);  
    outs << "Account balance $" << get_balance() << endl;
    outs << "Interest rate “" << get_rate() << “" << endl;
}
```

Display 6.7
(4/4)

Alternative BankAccount Class Implementation (part 4 of 4)

```cpp
// User faststream
void BankAccount::output(ostream & outs) {
    outs.setf(ios::fixed);
    outs.setf(ios::showpoint);
    outs.precision(2);  
    outs << "Account balance $" << get_balance() << endl;
    outs << "Interest rate “" << get_rate() << “" << endl;
}
```