Chapter 6
Structures and Classes
Learning Objectives

- **Structures**
  - Structure types
  - Structures as function arguments
  - Initializing structures

- **Classes**
  - Defining, member functions
  - Public and private members
  - Accessor and mutator functions
  - Structures vs. classes
Structures

- 2\textsuperscript{nd} aggregate data type: struct
- Recall: aggregate meaning "grouping"
  - Recall array: collection of values of same type
  - Structure: collection of values of different types
- Treated as a single item, like arrays
- Major difference: Must first "define" struct
  - Prior to declaring any variables
Structure Types

- Define struct globally (typically)
- No memory is allocated
  - Just a "placeholder" for what our struct will "look like"

- Definition:

```
struct CDAccountV1  // Name of new struct "type"
{
    double balance;  // member names
    double interestRate;
    int term;
};
```
Declare Structure Variable

- With structure type defined, now declare variables of this new type:

```c
CDAccountV1 account;
```
- Just like declaring simple types
- Variable `account` now of type `CDAccountV1`
- It contains "member values"
  - Each of the struct "parts"
Accessing Structure Members

- **Dot Operator to access members**
  - `account.balance`
  - `account.interestRate`
  - `account.term`

- **Called "member variables"**
  - The "parts" of the structure variable
  - Different structs can have the same name member variables
    - No conflicts
Display 6.1  A Structure Definition

1    //Program to demonstrate the CDAccountV1 structure type.
2    #include <iostream>
3    using namespace std;

4    //Structure for a bank certificate of deposit:
5    struct CDAccountV1
6    {
7        double balance;
8        double interestRate;
9        int term; //months until maturity
10    };

11   void getData(CDAccountV1& theAccount);
12   //Postcondition: theAccount.balance, theAccount.interestRate, and
13   //theAccount.term have been given values that the user entered at the keyboard.

An improved version of this structure will be given later in this chapter.
```c++
int main() {
    CDAccountV1 account;
    getData(account);

    double rateFraction, interest;
    rateFraction = account.interestRate/100.0;
    interest = account.balance*(rateFraction*(account.term/12.0));
    account.balance = account.balance + interest;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "When your CD matures in "
        << account.term << " months,\n"
        << "it will have a balance of $"
        << account.balance << endl;

    return 0;
}
```

(continued)
Display 6.1  A Structure Definition

```cpp
31   //Uses iostream:
32   void getData(CDAccountV1& theAccount)
33   {
34       cout < "Enter account balance: ";
35       cin >> theAccount.balance;
36       cout < "Enter account interest rate: ";
37       cin >> theAccount.interestRate;
38       cout < "Enter the number of months until maturity: ";
39       cin >> theAccount.term;
40   }
```

**SAMPLE DIALOGUE**

Enter account balance: $100.00
Enter account interest rate: 10.0
Enter the number of months until maturity: 6
When your CD matures in 6 months, it will have a balance of $105.00
Structure Pitfall

- Semicolon after structure definition
  - ; MUST exist:
    ```
    struct WeatherData
    {
        double temperature;
        double windVelocity;
    }; ← REQUIRED semicolon!
    ```
  - Required since you "can" declare structure variables in this location
Structure Assignments

- Given structure named CropYield

- Declare two structure variables:
  
  \[ \text{CropYield apples, oranges;} \]

  - Both are variables of "struct type CropYield"
  
  - Simple assignments are legal:
    
    \[ \text{apples = oranges;} \]
    
    - Simply copies each member variable from apples into member variables from oranges
Structures as Function Arguments

- Passed like any simple data type
  - Pass-by-value
  - Pass-by-reference
  - Or combination

- Can also be returned by function
  - Return-type is structure type
  - Return statement in function definition sends structure variable back to caller
Initializing Structures

- Can initialize at declaration
  
  - Example:
    ```c
    struct Date
    {
      int month;
      int day;
      int year;
    };
    
    Date dueDate = {12, 31, 2003};
    
    - Declaration provides initial data to all three member variables
    ```
Classes

- **Similar to structures**
  - Adds member **FUNCTIONS**
  - Not just member data

- **Integral to object-oriented programming**
  - Focus on objects
    - Object: Contains data and operations
    - In C++, variables of class type are objects
Class Definitions

- Defined similar to structures
- Example:
  ```c++
  class DayOfYear
  {
  public:
    void output();
    int month;
    int day;
  };
  ```
- Notice only member function’s prototype
  - Function’s implementation is elsewhere
Declaring Objects

- **Declared same as all variables**
  - Predefined types, structure types

- **Example:**
  ```
  DayOfYear today, birthday;
  ```
  - Declares two objects of class type DayOfYear

- **Objects include:**
  - Data
    - Members month, day
  - Operations (member functions)
    - output()
Class Member Access

- Members accessed same as structures
- Example:
  
  `today.month`
  
  `today.day`

  - And to access member function:
    
    `today.output();`  \(\rightarrow\)  Invokes member function
Class Member Functions

- Must define or "implement" class member functions
- Like other function definitions
  - Can be after main() definition
  - Must specify class:
    ```
    void DayOfYear::output()
    {...
    ```
    - `::` is scope resolution operator
    - Instructs compiler "what class" member is from
    - Item before `::` called type **qualifier**
Class Member Functions Definition

- Notice output() member function’s definition (in next example)
- Refers to member data of class
  - No qualifiers
- Function used for all objects of the class
  - Will refer to "that object’s" data when invoked
  - Example:
    ```
    today.output();
    - Displays "today" object’s data
    ```
Complete Class Example: Display 6.3 Class With a Member Function (1 of 4)

Display 6.3 Class with a Member Function

```cpp
// Program to demonstrate a very simple example of a class.
// A better version of the class DayOfYear will be given in Display 6.4.
#include <iostream>
using namespace std;

class DayOfYear
{
public:
    void output();
    int month;
    int day;
};

int main()
{
    DayOfYear today, birthday;
    cout << "Enter today's date:\n";
    cout << "Enter month as a number: ";
    cin >> today.month;
    cout << "Enter the day of the month: ";
    cin >> today.day;
    cout << "Enter your birthday:\n";
    cout << "Enter month as a number: ";
    cin >> birthday.month;
    cout << "Enter the day of the month: ";
    cin >> birthday.day;
}
```

Normally, member variables are **private** and not **public**, as in this example. This is discussed a bit later in this chapter.

Member function declaration

(continued)
Complete Class Example:
Display 6.3 Class With a Member Function (2 of 4)

Display 6.3  Class with a Member Function

```cpp
25    cout << "Today’s date is ";
26    today.output();
27    cout << endl;
28    cout << "Your birthday is ";
29    birthday.output();
30    cout << endl;
31    if (today.month == birthday.month && today.day == birthday.day)
32        cout << "Happy Birthday!\n";
33    else
34        cout << "Happy Unbirthday!\n";
35    return 0;
36 }
//Uses iostream:
37 void DayOfYear::output()
38 {
39     switch (month)
40     {
41         case 1:
42             cout << "January "; break;
43         case 2:
44             cout << "February "; break;
45         case 3:
46             cout << "March "; break;
47         case 4:
48             cout << "April "; break;
49     }
```

Calls to the member function output

Member function definition
Complete Class Example:
Display 6.3  Class With a Member Function (3 of 4)

```cpp
    case 5:
        cout << "May "; break;
    case 6:
        cout << "June "; break;
    case 7:
        cout << "July "; break;
    case 8:
        cout << "August "; break;
    case 9:
        cout << "September "; break;
    case 10:
        cout << "October "; break;
    case 11:
        cout << "November "; break;
    case 12:
        cout << "December "; break;
    default:
        cout << "Error in DayOfYear::output. Contact software vendor.";
    }
    cout << day;
```
Display 6.3  Class with a Member Function

**SAMPLE DIALOGUE**

Enter today's date:
Enter month as a number: 10
Enter the day of the month: 15
Enter your birthday:
Enter month as a number: 2
Enter the day of the month: 21
Today's date is October 15
Your birthday is February 21
Happy Unbirthday!
Dot and Scope Resolution Operator

- Used to specify "of what thing" they are members

- **Dot operator:**
  - Specifies member of particular object

- **Scope resolution operator:**
  - Specifies what class the function definition comes from
A Class’s Place

- **Class is full-fledged type!**
  - Just like data types int, double, etc.

- **Can have variables of a class type**
  - We simply call them "objects"

- **Can have parameters of a class type**
  - Pass-by-value
  - Pass-by-reference

- **Can use class type like any other type!**
Encapsulation

- Any data type includes
  - Data (range of data)
  - Operations (that can be performed on data)

- Example:
  - `int` data type has:
    - Data: -2147483648 to 2147483647 (for 32 bit int)
    - Operations: +,-,*,,/,%,logical,etc.

- Same with classes
  - But WE specify data, and the operations to be allowed on our data!
Abstract Data Types

- "Abstract"
  - Programmers don’t know details

- Abbreviated "ADT"
  - Collection of data values together with set of basic operations defined for the values

- ADT’s often "language-independent"
  - We implement ADT’s in C++ with classes
    - C++ class "defines" the ADT
  - Other languages implement ADT’s as well
More Encapsulation

- Encapsulation
  - Means "bringing together as one"

- Declare a class → get an object

- Object is "encapsulation" of
  - Data values
  - Operations on the data (member functions)
Principles of OOP

- **Information Hiding**
  - Details of how operations work not known to "user" of class

- **Data Abstraction**
  - Details of how data is manipulated within ADT/class not known to user

- **Encapsulation**
  - Bring together data and operations, but keep "details " hidden
Public and Private Members

- Data in class almost always designated private in definition!
  - Upholds principles of OOP
  - Hide data from user
  - Allow manipulation only via operations
    - Which are member functions

- Public items (usually member functions) are "user-accessible"
Modify previous example:

```cpp
class DayOfYear
{
    public:
        void input();
        void output();
    private:
        int month;
        int day;
};
```

- Data now private
- Objects have no direct access
Public and Private Example 2

- Given previous example
- Declare object:
  
  `DayOfYear today;`

- Object `today` can ONLY access public members
  
  - `cin >> today.month;` // NOT ALLOWED!
  - `cout << today.day;` // NOT ALLOWED!
  - Must instead call public operations:
    - `today.input();`
    - `today.output();`
Public and Private Style

- Can mix & match public & private

- More typically place public first
  - Allows easy viewing of portions that can be USED by programmers using the class
  - Private data is "hidden", so irrelevant to users

- Outside of class definition, cannot change (or even access) private data
Accessor and Mutator Functions

- **Object needs to "do something" with its data**

- **Call accessor member functions**
  - Allow object to read data
  - Also called "get member functions"
  - Simple retrieval of member data

- **Mutator member functions**
  - Allow object to change data
  - Manipulated based on application
Separate Interface and Implementation

- **User of class need not see details of how class is implemented**
  - Principle of OOP → encapsulation

- **User only needs "rules"**
  - Called "interface" for the class
    - In C++ → public member functions and associated comments

- **Implementation of class hidden**
  - Member function definitions elsewhere
  - User need not see them
Structures versus Classes

- **Structures**
  - Typically all members public
  - No member functions

- **Classes**
  - Typically all data members private
  - Interface member functions public

- **Technically, same**
  - Perceptually, very different mechanisms
Thinking Objects

- **Focus for programming changes**
  - Before → algorithms center stage
  - OOP → data is focus

- **Algorithms still exist**
  - They simply focus on their data
  - Are "made" to "fit" the data

- **Designing software solution**
  - Define variety of objects and how they interact