



# 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<sup>nd</sup> 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:  
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 same name member variables
    - No conflicts



# Structure Example

Display 6.1,  
page 225

Display 6.1 A Structure Definition (part 1 of 2)

```
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.

14 int main()
15 {
16     CDAccountV1 account;
17     getData(account);

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

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

29     return 0;
30 }
```

*An improved version of this structure will be given later in this chapter.*



# Structure Example Cont'd

## Display 6.1, page 225

### Display 6.1 A Structure Definition (*part 2 of 2*)

---

```
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:  
CropYield apples, oranges;
  - Both are variables of 'struct type CropYield'
  - Simple assignments are legal:  
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:

```
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:

```
class DayOfYear ← name of new class type
{
public:
    void output(); ← member function!
    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(); ← 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,  
page 237

Display 6.3 Class with a Member Function (part 1 of 2)

```
1 //Program to demonstrate a very simple example of a class.
2 //A better version of the class DayOfYear will be given in Display 6.4.
3 #include <iostream>
4 using namespace std;
5 class DayOfYear
6 {
7 public:
8     void output( );
9     int month;
10    int day;
11 };
12 int main( )
13 {
14     DayOfYear today, birthday;
15     cout << "Enter today's date:\n";
16     cout << "Enter month as a number: ";
17     cin >> today.month;
18     cout << "Enter the day of the month: ";
19     cin >> today.day;
20     cout << "Enter your birthday:\n";
21     cout << "Enter month as a number: ";
22     cin >> birthday.month;
23     cout << "Enter the day of the month: ";
24     cin >> birthday.day;
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 }
```

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

*Member function declaration*

*Calls to the member function output*



# Complete Class Example Cont'd

## Display 6.3, page 237

### Display 6.1 A Structure Definition (*part 2 of 2*)

---

```
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
```



# 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: +-32,767
    - 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'



# Public and Private Example

- **Modify previous example:**

```
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 it's 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
  - Perceptionally, 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



# Summary 1

- Structure is collection of different types
- Class used to combine data and functions into single unit -> object
- Member variables and member functions
  - Can be public → accessed outside class
  - Can be private → accessed only in a member function's definition
- Class and structure types can be formal parameters to functions



# Summary 2

- C++ class definition
  - Should separate two key parts
    - Interface: what user needs
    - Implementation: details of how class works