

Structures, classes and files

Structures

- A Struct is a user-defined data type for holding data
- 2nd 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 A Structure Definition (1 of 3)

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 keyboar
```

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

Structure Example:

Display 6.1 A Structure Definition (2 of 3)

```
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  }
```

(continued)

Structure Example:

Display 6.1 A Structure Definition (3 of 3)

Display 6.1 A Structure Definition

```
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 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 and Objects

- A Class is a user-defined data type for holding data and functions
- Classes are declared using the keyword **class**

```
class class_name{
```

```
    access_specifier1:
```

```
        member1;
```

```
    access_specifier2:
```

```
        member2;
```

```
}
```

An access-specifier is one of the following three keywords:
private, public, protected

- An object is an instantiation of a class

```
int number1;
```

```
class_name object_name;
```

data type (pointing to **class_name**)
variable (pointing to **object_name**)

Example: `cout` *is an object of class* `ostream`

Class Example: gradeBook1.cpp

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

```
class GradeBook{
public:
```

```
    void displayMessage() {
```

```
        cout << "Welcome to the Grade Book!" << endl;
```

```
    }
```

```
};
```

```
int main() {
```

```
    GradeBook myGradeBook;
```

```
    myGradeBook.displayMessage();
```

```
    return 0;
```

```
}
```

*Note: Class definition begins with the keyword **class** and ends with a semi-colon. It contains a member function.*

Name of the class: GradeBook

Name of the object: myGradeBook

Output:

Welcome to the Grade Book!

Class Example: gradeBook2.cpp (1)

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

class GradeBook{
public:
    void displayMessage(string nameOfCourse){
        cout <<"Welcome to Grade Book for " << nameOfCourse << "!\n";
    }
};

int main(){
    string nameOfCourse;
    GradeBook myGradeBook;
    cout << "Enter the course name" << endl;
    getline(cin, nameOfCourse);
    myGradeBook.displayMessage(nameOfCourse);
    return 0;
}
```

Class Example: gradeBook2.cpp (2)

Output:

```
Enter the course name
```

```
CS101 Introduction to C++
```

```
Welcome to the Grade Book for CS101 Introduction to C++!
```

Note:

To obtain the course name, we did not use

```
cin >> nameOfCourse;
```

This is because reads the input until the first white-space character is reached.

Thus `cin` will only read `CS101`. Therefore we used the following function that reads the input stream till it encounters a newline character:

```
getline(cin, nameOfCourse);
```

Notes Regarding Access-Specifiers

- **public** members can be accessed from outside the class also
- **private** data members can be only accessed from within the class
- **protected** data members can be accessed by a class and its subclass
- By default, access-specifier is **private**

Constructor & Destructor

- Every time an instance of a class is created the constructor method is called
- The constructor has the same name as the class and it doesn't return any type
- The destructor's name is defined in the same way as a constructor, but with a '~' in front
- The compiler provides a default constructor if none is specified in the program

Constructor & Destructor: constDest.cpp (1)

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

class Point{
public:
    int x;
    int y;

    Point() {
        cout << "Default Constructor" << endl;
    }

    ~Point() {
        cout << "Default Destructor" << endl;
    }

};

. . .
```

Constructor & Destructor: constDest.cpp

(2)

```
. . .  
int main(){  
    Point p;  
    p.x = 10;  
    p.y = 20;  
    cout << "Value of class varibales x and y: ";  
    cout << p.x << ", " << p.y;  
    cout << endl;  
    return 0;  
}
```

Output:

Default Constructor

Value of class varibales x and y: 10, 20

Default Destructor

Class Example: gradeBook3.cpp

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

class GradeBook{
public:
    void displayMessage(string nameOfCourse);
};

int main(){
    string nameOfCourse;
    GradeBook myGradeBook;
    cout << "Enter the course name" << endl;
    getline(cin, nameOfCourse);
    myGradeBook.displayMessage(nameOfCourse);
    return 0;
}

void GradeBook::displayMessage(string nameOfCourse){
    cout << "Welcome to Grade Book for " << nameOfCourse << "!\n";
}
```

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

Constructor Definitions

- Constructors defined like any member function
 - Except:
 1. Must have same name as class
 2. Cannot return a value; not even void!

Constructor Definition Example

- Class definition with constructor:
 - class DayOfYear
 {
 public:
 DayOfYear(int monthValue, int dayValue);
 //Constructor initializes month and day
 void input();
 void output();
 ...
 private:
 int month;
 int day;
 }

Constructor Notes

- Notice name of constructor: DayOfYear
 - Same name as class itself!
- Constructor declaration has no return-type
 - Not even void!
- Constructor in public section
 - It's called when objects are declared
 - If private, could never declare objects!

Calling Constructors

- Declare objects:
 DayOfYear date1(7, 4),
 date2(5, 5);
- Objects are created here
 - Constructor is called
 - Values in parens passed as arguments to constructor
 - Member variables month, day initialized:
 date1.month → 7 date2.month → 5
 date1.day → 4 date2.day → 5

Constructor Equivalency

- Consider:
 - DayOfYear date1, date2
date1.DayOfYear(7, 4); // ILLEGAL!
date2.DayOfYear(5, 5); // ILLEGAL!
- Seemingly OK...
 - CANNOT call constructors like other member functions!

Constructor Code

- Constructor definition is like all other member functions:

```
DayOfYear::DayOfYear(int monthValue, int dayValue)
{
    month = monthValue;
    day = dayValue;
}
```

- Note same name around ::
 - Clearly identifies a constructor
- Note no return type
 - Just as in class definition

Alternative Definition

- Previous definition equivalent to:

```
DayOfYear::DayOfYear(          int monthValue,  
                             int dayValue)  
    : month(monthValue), day(dayValue) ←  
{...}
```

- Third line called "Initialization Section"
- Body left empty
- Preferable definition version

Overloaded Constructors

- Can overload constructors just like other functions
- Recall: a signature consists of:
 - Name of function
 - Parameter list
- Provide constructors for all possible argument-lists
 - Particularly "how many"

Class with Constructors Example:

Display 7.1 Class with Constructors (1 of 3)

Display 7.1 Class with Constructors

```
1  #include <iostream>
2  #include <cstdlib> //for exit
3  using namespace std;

4  class DayOfYear
5  {
6  public:
7      DayOfYear(int monthValue, int dayValue);
8      //Initializes the month and day to arguments.

9      DayOfYear(int monthValue);
10     //Initializes the date to the first of the given month.

11     DayOfYear( ); ← default constructor
12     //Initializes the date to January 1.

13     void input( );
14     void output( );
15     int getMonthNumber( );
16     //Returns 1 for January, 2 for February, etc.
```

This definition of DayOfYear is an improved version of the class DayOfYear given in Display 6.4.

Class with Constructors Example:


Display 7.1 Class with Constructors (2 of 3)

```
17     int getDay( );
18 private:
19     int month;
20     int day;
21     void testDate( );
22 };
```


```
23 int main( )
24 {
25     DayOfYear date1(2, 21), date2(5), date3;
26     cout << "Initialized dates:\n";
27     date1.output( ); cout << endl;
28     date2.output( ); cout << endl;
29     date3.output( ); cout << endl;
30
31     date1 = DayOfYear(10, 31);
32     cout << "date1 reset to the following:\n";
33     date1.output( ); cout << endl;
34     return 0;
35 }
```

```
36 DayOfYear::DayOfYear(int monthValue, int dayValue)
37     : month(monthValue), day(dayValue)
38 {
39     testDate( );
40 }
```

This causes a call to the default constructor. Notice that there are no parentheses.



an explicit call to the constructor
DayOfYear::DayOfYear



Class with Constructors Example:

Display 7.1 Class with Constructors (3 of 3)

Display 7.1 Class with Constructors

```
41 DayOfYear::DayOfYear(int monthValue) : month(monthValue), day(1)
42 {
43     testDate( );
44 }

45 DayOfYear::DayOfYear( ) : month(1), day(1)
46 { /*Body intentionally empty.*/}

47 //uses iostream and cstdlib:
48 void DayOfYear::testDate( )
49 {
50     if ((month < 1) || (month > 12))
51     {
52         cout << "Illegal month value!\n";
53         exit(1);
54     }
55     if ((day < 1) || (day > 31))
56     {
57         cout << "Illegal day value!\n";
58         exit(1);
59     }
60 }
```

<Definitions of the other member functions are the same as in Display 6.4.>

SAMPLE DIALOGUE

Initialized dates:
February 21
May 1
January 1
date1 reset to the following:
October 31

Constructor with No Arguments

- Can be confusing
- Standard functions with no arguments:
 - Called with syntax: `callMyFunction();`
 - Including empty parentheses
- Object declarations with no "initializers":
 - `DayOfYear date1; // This way!`
 - `DayOfYear date(); // NO!`
 - What is this really?
 - Compiler sees a function declaration/prototype!
 - Yes! Look closely!

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

- Example

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

- Data now private
- Objects have no direct access
 - Use accessor and mutator functions

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

File I/O

- C++ provides the following classes to perform output and input of characters to/from files:

ofstream: Stream class to write on files

ifstream: Stream class to read from files

fstream: Stream class to both read and write from/to files.

- Objects of these classes are associated to a real file by opening a file as:
- `open (filename, mode);`

Modes of Files

- Mode is an optional parameter with a combination of the following flags

`ios::in` Open for input operations

`ios::out` Open for output operations

`ios::app` All input operations are performed at the end of the file (i.e. append more data)

- there are few more flags:
- More information:
- <http://www.cplusplus.com/doc/tutorial/files/>

Write to a file: fileWrite.cpp

```
#include <iostream>
```

```
#include <fstream>
```

*Stream class to both
read and write from/to
files*

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    ofstream myfile, myfile2;
```

```
    myfile.open ("example.txt");
```

```
    myfile << "Writing this to a file.\n";
```

```
    myfile.close();
```

```
    myfile2.open ("example.txt",ios::app);
```

```
    myfile2 << "Appending 2nd line this to same file.\n";
```

```
    myfile2.close();
```

```
    return 0;
```

```
}
```

*Two ofstream objects created
Notice that the mode in which the file
should be opened is not specified.
Default mode is ios::out when ofstream
object is used*

*file is opened under the append
mode*

This code creates a file called example.txt and inserts two sentences into it in the same way we are used to do with cout, but using the file stream myfile instead.

Reading From File & Writing to Console: fileReadScreenWrite.cpp

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main () {
    string line;
    ifstream myfile ("example.txt");
    if (myfile.is_open()){
        while ( myfile.good() )
        {
            getline (myfile,line);
            cout << line << endl;
        }
        myfile.close();
    }
    else
        cout << "Unable to open file";
    return 0;
}
```

The function `myfile.good()` will return true in the case the stream is ready for input/output operations, false when end of file is reached