Chapter 6: Classes and Data Abstraction

Outline

6.1 Introduction
6.2 Structure Definitions
6.3 Accessing Members of Structures
6.4 Implementing a User-Defined Type Time with a Struct
6.5 Implementing a Time Abstract Data Type with a Class
6.6 Class Scope and Accessing Class Members
6.7 Separating Interface from Implementation
6.8 Controlling Access to Members
6.9 Access Functions and Utility Functions
6.10 Initializing Class Objects: Constructors
6.11 Using Default Arguments with Constructors
6.12 Using Destructors
6.13 When Constructors and Destructors Are Called
6.14 Using Data Members and Member Functions
6.15 A Subtle Trap: Returning a Reference to a Private Data Member
6.16 Assignment by Default Memberwise Copy
6.17 Software Reusability
6.1 Introduction

- **Object-oriented programming (OOP)**
  - Encapsulates data (attributes) and functions (behavior) into packages called classes
- **Information hiding**
  - Implementation details are hidden within the classes themselves
- **Classes**
  - Classes are the standard unit of programming
  - A class is like a blueprint – reusable
  - Objects are instantiated (created) from the class
  - For example, a house is an instance of a “blueprint class”
6.2 Structure Definitions

- **Structures**
  - Aggregate data types built using elements of other types

```c
struct Time {
    int hour;
    int minute;
    int second;
};
```

- Members of the same structure must have unique names
- Two different structures may contain members of the same name
- Each structure definition must end with a semicolon
6.2 Structure Definitions

• Self-referential structure
  – Contains a member that is a pointer to the same structure type
  – Used for linked lists, queues, stacks and trees

• `struct`
  – Creates a new data type that is used to declare variables
  – Structure variables are declared like variables of other types
  – Example:

    ```
    Time timeObject, timeArray[10],
    *timePtr, &timeRef = timeObject;
    ```
6.3 Accessing Members of Structures

• Member access operators:
  – Dot operator ( . ) for structures and objects
  – Arrow operator ( -> ) for pointers
  – Print member hour of timeObject:
    
    ```cpp
    cout << timeObject.hour;
    ```
    
    OR
    
    ```cpp
    timePtr = &timeObject;
    cout << timePtr->hour;
    ```
  – timePtr->hour is the same as ( *timePtr ).hour
  – Parentheses required: * has lower precedence than .
// Fig. 6.1: fig06_01.cpp
// Create a structure, set its members, and print it.
#include <iostream>

using std::cout;
using std::endl;

struct Time { // structure definition
    int hour;     // 0-23
    int minute;   // 0-59
    int second;   // 0-59
};

void printMilitary( const Time & ); // prototype
void printStandard( const Time & );  // prototype

int main()
{
    Time dinnerTime; // variable of new type Time

    // set members to valid values
    dinnerTime.hour = 18;
    dinnerTime.minute = 30;
    dinnerTime.second = 0;

    cout << "Dinner will be held at ";
    printMilitary( dinnerTime );
    cout << " military time,\nwhich is ";
    printStandard( dinnerTime );
    cout << " standard time.\n";

    return 0;
}
2.2 Set the time to an invalid hour, then print it

3. Define the functions printMilitary and printStandard

```cpp
// set members to invalid values
dinnerTime.hour = 29;
dinnerTime.minute = 73;

cout << "\nTime with invalid values: ";
printMilitary( dinnerTime );
cout << endl;
return 0;

// Print the time in military format
void printMilitary( const Time &t )
{
    cout << ( t.hour < 10 ? "0" : "" ) << t.hour << ":";
    << ( t.minute < 10 ? "0" : "" ) << t.minute;
}

// Print the time in standard format
void printStandard( const Time &t )
{
    cout << ( ( t.hour == 0 || t.hour == 12 ) ?
             12 : t.hour % 12 )
         << ":" << ( t.minute < 10 ? "0" : "" ) << t.minute
         << ":" << ( t.second < 10 ? "0" : "" ) << t.second
         << ( t.hour < 12 ? " AM" : " PM" );
}"
```
Dinner will be held at 18:30 military time, which is 6:30:00 PM standard time.

Time with invalid values: 29:73
6.5 Implementing a Time Abstract Data Type with a Class

• Classes
  – Model objects that have attributes (data members) and behaviors (member functions)
  – Defined using keyword `class`
  – Have a body delineated with braces ( `{` and `}`)
  – Class definitions terminate with a semicolon
  – Example:

    ```cpp
    class Time {
    public:
        Time();
        void setTime( int, int, int );
        void printMilitary();
        void printStandard();
    private:
        int hour;     // 0 - 23
        int minute;   // 0 - 59
        int second;   // 0 - 59
    }; 
    ```

    **Public:** and **Private:** are member-access specifiers.

    **setTime, printMilitary, and printStandard** are member functions.

    **Time** is the constructor.

    **hour, minute, and second** are data members.
6.5 Implementing a Time Abstract Data Type with a Class

• Member access specifiers
  – Classes can limit the access to their member functions and data
  – The three types of access a class can grant are:
    • Public — Accessible wherever the program has access to an object of the class
    • private — Accessible only to member functions of the class
    • Protected — Similar to private and discussed later

• Constructor
  – Special member function that initializes the data members of a class object
  – Cannot return values
  – Have the same name as the class
6.5 Implementing a Time Abstract Data Type with a Class

• Class definition and declaration
  – Once a class has been defined, it can be used as a type in object, array and pointer declarations
  – Example:

```cpp
Time sunset,                  // object of type Time
arrayOfTimes[ 5 ],          // array of Time objects
*pointerToTime,              // pointer to a Time object
&dinnerTime = sunset;        // reference to a Time object
```

Note: The class name becomes the new type specifier.
// Fig. 6.3: fig06_03.cpp
// Time class.
#include <iostream>

using std::cout;
using std::endl;

// Time abstract data type (ADT) definition
class Time {
public:
    Time();                        // constructor
    void setTime( int, int, int ); // set hour, minute, second
    void printMilitary();          // print military time format
    void printStandard();          // print standard time format
private:
    int hour;     // 0 – 23
    int minute;   // 0 – 59
    int second;   // 0 – 59
};

// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time() { hour = minute = second = 0; }

// Set a new Time value using military time. Perform validity
// checks on the data values. Set invalid values to zero.
void Time::setTime( int h, int m, int s )
{
    hour = ( h >= 0 && h < 24 ) ? h : 0;
    minute = ( m >= 0 && m < 60 ) ? m : 0;
    second = ( s >= 0 && s < 60 ) ? s : 0;
}
// Print Time in military format
void Time::printMilitary()
{
    cout << ( hour < 10 ? "0" : "" ) << hour << ":" << ( minute < 10 ? "0" : "" ) << minute;
}

// Print Time in standard format
void Time::printStandard()
{
    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":" << ( minute < 10 ? "0" : "" ) << minute
    << ":" << ( second < 10 ? "0" : "" ) << second
    << ( hour < 12 ? " AM" : " PM" );
}

// Driver to test simple class Time
int main()
{
    Time t; // instantiate object t of class Time
    cout << "The initial military time is ";
    t.printMilitary();
    cout << "The initial standard time is ";
    t.printStandard();

    The initial military time is 00:00
    The initial standard time is 12:00:00 AM

    Notice how functions are called using the dot (.) operator.
The initial military time is 00:00
The initial standard time is 12:00:00 AM

Military time after setTime is 13:27
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Military time: 00:00
Standard time: 12:00:00 AM
6.5 Implementing a Time Abstract Data Type with a Class

• Destructors
  – Functions with the same name as the class but preceded with a tilde character (~)
  – Cannot take arguments and cannot be overloaded
  – Performs “termination housekeeping”

• Binary scope resolution operator (::)
  – Combines the class name with the member function name
  – Different classes can have member functions with the same name

• Format for defining member functions

```
ReturnType ClassName::MemberFunctionName( ){

    ...

}
```
6.5 Implementing a Time Abstract Data Type with a Class

• If a member function is defined inside the class
  – Scope resolution operator and class name are not needed
  – Defining a function outside a class does not change it being public or private

• Classes encourage software reuse
  – Inheritance allows new classes to be derived from old ones
6.6 Class Scope and Accessing Class Members

- **Class scope**
  - Data members and member functions

- **File scope**
  - Non member functions

- **Inside a scope**
  - Members accessible by all member functions
    - Referenced by name

- **Outside a scope**
  - Members are referenced through handles
    - An object name, a reference to an object or a pointer to an object
6.6 Class Scope and Accessing Class Members

• Function scope
  – Variables only known to function they are defined in
  – Variables are destroyed after function completion

• Accessing class members
  – Same as structs
  – Dot (.) for objects and arrow (->) for pointers
  – Example:
    • `t.hour` is the `hour` element of `t`
    • `TimePtr->hour` is the `hour` element
// Fig. 6.4: fig06_04.cpp
// Demonstrating the class member access operators . and ->
//
// CAUTION: IN FUTURE EXAMPLES WE AVOID PUBLIC DATA!
#include <iostream>

using std::cout;
using std::endl;

// Simple class Count
class Count {
  public:
    int x;
    void print() { cout << x << endl; }
};

int main()
{
    Count counter,          // create counter object
        *counterPtr = &counter, // pointer to counter
        &counterRef = counter; // reference to counter

    cout << "Assign 7 to x and print using the object's name: ";
    counter.x = 7;          // assign 7 to data member x
    counter.print();        // call member function print

    cout << "Assign 8 to x and print using a reference: ";
    counterRef.x = 8;       // assign 8 to data member x
    counterRef.print();     // call member function print

    It is rare to have public member variables. Usually only member functions are public; this keeps as much information hidden as possible.
Assign 7 to x and print using the object's name: 7
Assign 8 to x and print using a reference: 8
Assign 10 to x and print using a pointer: 10

```cpp
31    cout << "Assign 10 to x and print using a pointer: ";
32    counterPtr->x = 10;  // assign 10 to data member x
33    counterPtr->print(); // call member function print
34    return 0;
35 }
```
6.7 Separating Interface from Implementation

• Separating interface from implementation
  – Makes it easier to modify programs
  – Header files
    • Contains class definitions and function prototypes
  – Source-code files
    • Contains member function definitions
// Fig. 6.5: time1.h
// Declaration of the Time class.
// Member functions are defined in time1.cpp

// prevent multiple inclusions of header file
#ifndef TIME1_H
#define TIME1_H

// Time abstract data type definition
class Time {
 public:
  Time();                        // constructor
  void setTime( int, int, int ); // set hour, minute, second
  void printMilitary();          // print military time format
  void printStandard();          // print standard time format

 private:
  int hour;     // 0 - 23
  int minute;   // 0 - 59
  int second;   // 0 - 59
};
#endif

If time1.h (TIME1_H) is not defined (#ifndef) then it is loaded (#define TIME1_H). If TIME1_H is already defined, then everything up to #endif is ignored.
This prevents loading a header file multiple times.

Dot (.) replaced with underscore (_) in file name.
// Fig. 6.5: time1.cpp
// Member function definitions for Time class.
#include <iostream>

using std::cout;

#include "time1.h"

// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time() { hour = minute = second = 0; }

// Set a new Time value using military time. Perform validity
// checks on the data values. Set invalid values to zero.
void Time::setTime( int h, int m, int s )
{
    hour   = ( h >= 0 && h < 24 ) ? h : 0;
    minute = ( m >= 0 && m < 60 ) ? m : 0;
    second = ( s >= 0 && s < 60 ) ? s : 0;
}

// Print Time in military format
void Time::printMilitary()
{
    cout << ( hour < 10 ? "0" : "" ) << hour << ":" << ( minute < 10 ? "0" : "" ) << minute;
}

// Print time in standard format
void Time::printStandard()
{
    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
         << ":" << ( minute < 10 ? "0" : "" ) << minute
         << ":" << ( second < 10 ? "0" : "" ) << second
         << ( hour < 12 ? " AM" : " PM" );
}
6.8 Controlling Access to Members

• **public**
  - Presents clients with a view of the services the class provides (interface)
  - Data and member functions are accessible

• **private**
  - Default access mode
  - Data only accessible to member functions and friends
  - **private** members only accessible through the **public** class interface using **public** member functions
1 // Fig. 6.6: fig06_06.cpp
2 // Demonstrate errors resulting from attempts
3 // to access private class members.
4 #include <iostream>
5
6 using std::cout;
7
8 #include "time1.h"
9
10 int main()
11 {
12    Time t;
13
14    // Error: 'Time::hour' is not accessible
15    t.hour = 7;
16
17    // Error: 'Time::minute' is not accessible
18    cout << "minute = " << t.minute;
19
20    return 0;
21 }

Compiling...
Fig06_06.cpp
D:\Fig06_06.cpp(15) : error C2248: 'hour' : cannot access private member declared in class 'Time'
D:\Fig6_06\time1.h(18) : see declaration of 'hour'
D:\Fig06_06.cpp(18) : error C2248: 'minute' : cannot access private member declared in class 'Time'
D:\time1.h(19) : see declaration of 'minute'
Error executing cl.exe.

test.exe - 2 error(s), 0 warning(s)
6.9 Access Functions and Utility Functions

• Utility functions
  – *private* functions that support the operation of public functions
  – Not intended to be used directly by clients

• Access functions
  – *public* functions that read/display data or check conditions
  – Allow *public* functions to check *private* data

• Following example
  – Program to take in monthly sales and output the total
  – Implementation not shown, only access functions
// Fig. 6.7: fig06_07.cpp
// Demonstrating a utility function
// Compile with salesp.cpp
#include "salesp.h"

int main()
{
    SalesPerson s;         // create SalesPerson object s
    s.getSalesFromUser();  // note simple sequential code
    s.printAnnualSales();  // no control structures in main
    return 0;
}

OUTPUT
Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

The total annual sales are: $60120.59

Create object s, an instance of class SalesPerson

1. Load header file and compile with the file that contains the function definitions
2. Create an object

2.1 Use the object’s member functions to

Use access functions to gather and print data (getSalesFromUser and printAnnualSales).
Utility functions actually calculate the total sales, but the user is not aware of these function calls.

Notice how simple main() is – there are no control structures, only function calls. This hides the implementation of the program.
6.10 Initializing Class Objects: Constructors

• Constructors
  – Initialize class members
  – Same name as the class
  – No return type
  – Member variables can be initialized by the constructor or set afterwards

• Passing arguments to a constructor
  – When an object of a class is declared, initializers can be provided
  – Format of declaration with initializers:
    \[\text{Class-type ObjectName( value1,value2,…);}\]
  – Default arguments may also be specified in the constructor prototype
// Fig. 6.8: time2.h
// Declaration of the Time class.
// Member functions are defined in time2.cpp

// preprocessor directives that
// prevent multiple inclusions of header file
#ifndef TIME2_H
#define TIME2_H

// Time abstract data type definition
class Time {

public:
    Time( int = 0, int = 0, int = 0 ); // default constructor
    void setTime( int, int, int ); // set hour, minute, second
    void printMilitary(); // print military time format
    void printStandard(); // print standard time format

private:
    int hour; // 0 - 23
    int minute; // 0 - 59
    int second; // 0 - 59
};

#endif

Notice that default settings for the three member variables are set in constructor prototype. No names are needed; the defaults are applied in the order the member variables are declared.
// Fig. 6.8: fig06_08.cpp
// Demonstrating a default constructor
// function for class Time.
#include <iostream>

using std::cout;
using std::endl;

#include "time2.h"

int main()
{
    Time t1,             // all arguments defaulted
    t2(2),          // minute and second defaulted
    t3(21, 34),     // second defaulted
    t4(12, 25, 42), // all values specified
    t5(27, 74, 99); // all bad values specified

    cout << "Constructed with:
    " << "all arguments defaulted:
   " << t1.printMilitary();
    cout << t1.printStandard();

    cout << "hour specified; minute and second defaulted:" << "
   " << t2.printMilitary();
    cout << t2.printStandard();

    cout << "hour and minute specified; second defaulted:" << "
   " << t3.printMilitary();
}


```cpp
94 cout << "\n ";
95 t3.printStandard();
96 cout << "\n ";
97 cout << "\nhour, minute, and second specified:" << "\n ";
98 t4.printMilitary();
99 cout << "\n ";
100 cout << "\n ";
101 t4.printStandard();
102 cout << "\nall invalid values specified:" << "\n ";
103 t5.printMilitary();
104 cout << "\n ";
105 t5.printStandard();
106 cout << endl;
107 return 0;
111}
```

**OUTPUT**

Constructed with:
- all arguments defaulted:
  - 00:00
  - 12:00:00 AM
- hour specified; minute and second defaulted:
  - 02:00
  - 2:00:00 AM
- hour and minute specified; second defaulted:
  - 21:34
  - 9:34:00 PM
- hour, minute, and second specified:
  - 12:25
  - 12:25:42 PM
- all invalid values specified:
  - 00:00
  - 12:00:00 AM

When only **hour** is specified, **minute** and **second** are set to their default values of **0**.
6.12 Using Destructors

- **Destructors**
  - Are member function of class
  - Perform termination housekeeping before the system reclaims the object’s memory
  - Complement of the constructor
  - Name is tilde (~) followed by the class name (i.e., ~\texttt{Time})
    - Recall that the constructor’s name is the class name
  - Receives no parameters, returns no value
  - One destructor per class
    - No overloading allowed
6.13 When Constructors and Destructors Are Called

• Constructors and destructors called automatically
  – Order depends on scope of objects

• Global scope objects
  – Constructors called before any other function (including `main`)
  – Destructors called when `main` terminates (or `exit` function called)
  – Destructors not called if program terminates with `abort`

• Automatic local objects
  – Constructors called when objects are defined
  – Destructors called when objects leave scope
    • i.e., when the block in which they are defined is exited
  – Destructors not called if the program ends with `exit` or `abort`
6.13 When Constructors and Destructors Are Called

• Static local objects
  – Constructors called when execution reaches the point where the objects are defined
  – Destructors called when `main` terminates or the `exit` function is called
  – Destructors not called if the program ends with `abort`
1 // Fig. 6.9: create.h
2 // Definition of class CreateAndDestroy.
3 // Member functions defined in create.cpp.
4 #ifndef CREATE_H
5 #define CREATE_H
6
7 class CreateAndDestroy {
8    public:
9        CreateAndDestroy( int );  // constructor
10        ~CreateAndDestroy();    // destructor
11    private:
12        int data;
13    }
14
15 #endif
16 // Fig. 6.9: create.cpp
17 // Member function definitions for class CreateAndDestroy
18 #include <iostream>
19
20 using std::cout;
21 using std::endl;
22
23 #include "create.h"
24
25 CreateAndDestroy::CreateAndDestroy( int value )
26 {
27    data = value;
28    cout << "Object " << data << "    constructor";
29 }
30
31 CreateAndDestroy::~CreateAndDestroy()
32 { cout << "Object " << data << "    destructor " << endl; }
3. Create multiple objects of varying types

```cpp
// Fig. 6.9: fig06_09.cpp
// Demonstrating the order in which constructors and destructors are called.
#include <iostream>

using std::cout;
using std::endl;

#include "create.h"

void create( void );   // prototype

CreateAndDestroy first( 1 );  // global object

int main()
{
    cout << " (global created before main)" << endl;
    CreateAndDestroy second( 2 );        // local object
    cout << " (local automatic in main)" << endl;
    static CreateAndDestroy third( 3 );  // local object
    cout << " (local static in main)" << endl;
    create();  // call function to create objects
    CreateAndDestroy fourth( 4 );        // local object
    cout << " (local automatic in main)" << endl;
    return 0;
}
```
// Function to create objects
void create( void )
{
    CreateAndDestroy fifth( 5 );
    cout << "   (local automatic in create)" << endl;

    static CreateAndDestroy sixth( 6 );
    cout << "   (local static in create)" << endl;

    CreateAndDestroy seventh( 7 );
    cout << "   (local automatic in create)" << endl;
}

OUTPUT
Object 1   constructor   (global created before main)
Object 2   constructor   (local automatic in main)
Object 3   constructor   (local static in main)
Object 5   constructor   (local automatic in create)
Object 6   constructor   (local static in create)
Object 7   constructor   (local automatic in create)
Object 7   destructor
Object 5   destructor
Object 4   constructor   (local automatic in main)
Object 4   destructor
Object 2   destructor
Object 6   destructor
Object 3   destructor
Object 1   destructor

Notice how the order of the constructor and destructor call depends on the types of variables (automatic, global and static) they are associated with.
6.14 Using Data Members and Member Functions

• Member functions
  – Allow clients of the class to *set* (i.e., write) or *get* (i.e., read) the values of private data members
  – Example:
    
    *Adjusting a customer’s bank balance*

    • *private* data member *balance* of a class *BankAccount* could be modified through the use of member function *computeInterest*

    • A member function that sets data member *interestRate* could be called *setInterestRate*, and a member function that returns the *interestRate* could be called *getInterestRate*

    – Providing *set* and *get* functions does not make *private* variables *public*

    – A set function should ensure that the new value is valid
6.15 A Subtle Trap: Returning a Reference to a Private Data Member

• Reference to an object
  – Alias for the name of the object
  – May be used on the left side of an assignment statement
  – Reference can receive a value, which changes the original object as well

• Returning references
  – public member functions can return non-const references to private data members
    • Should be avoided, breaks encapsulation
1 // Fig. 6.11: time4.h
2 // Declaration of the Time class.
3 // Member functions defined in time4.cpp
4
5 // preprocessor directives that
6 // prevent multiple inclusions of header file
7 ifndef TIME4_H
8 #define TIME4_H
9
10 class Time {
11 public:
12    Time( int = 0, int = 0, int = 0 );
13    void setTime( int, int, int );
14    int getHour();
15    int &badSetHour( int );  // DANGEROUS reference return
16 private:
17    int hour;
18    int minute;
19    int second;
20};
21
22 endif

Notice how member function
badSetHour returns a reference
(int & is the return type).
// Fig. 6.11: time4.cpp
// Member function definitions for Time class.
#include "time4.h"

// Constructor function to initialize private data.
// Calls member function setTime to set variables.
// Default values are 0 (see class definition).
Time::Time( int hr, int min, int sec )
{ setTime( hr, min, sec ); }

// Set the values of hour, minute, and second.
void Time::setTime( int h, int m, int s )
{
    hour   = ( h >= 0 && h < 24 ) ? h : 0;
    minute = ( m >= 0 && m < 60 ) ? m : 0;
    second = ( s >= 0 && s < 60 ) ? s : 0;
}

// Get the hour value
int Time::getHour() { return hour; }

// POOR PROGRAMMING PRACTICE:
// Returning a reference to a private data member.
int &Time::badSetHour( int hh )
{
    hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
    return hour;  // DANGEROUS reference return
}

badSetHour returns a reference to the private member variable hour. Changing this reference will alter hour as well.
Declare Time object \texttt{t} and reference \texttt{hourRef} that is assigned the reference returned by the call \texttt{t.badSetHour(20)}.

Hour before modification: 20

Alias used to set the value of \texttt{hour} to 30 (an invalid value).

Hour after modification: 30

Function call used as an \textit{lvalue} and assigned the value 74 (another invalid value).

POOR PROGRAMMING PRACTICE!!!!!!!
badSetHour as an lvalue, Hour: 74

*********************************
POOR PROGRAMMING PRACTICE!!!!!!!
badSetHour as an lvalue, Hour: 74

*********************************
HourRef used to change hour to an invalid value. Normally, the function setbadsethour would not have allowed this. However, because it returned a reference, hour was changed directly.

POOR PROGRAMMING PRACTICE!!!!!!!
badsethour as an lvalue, Hour: 74
*********************************
Hour before modification: 20
Hour after modification: 30

*********************************
6.16 Assignment by Default Memberwise Copy

• Assigning objects
  – An object can be assigned to another object of the same type using the assignment operator (≡)
  – Member by member copy

• Objects may be
  – Passed as function arguments
  – Returned from functions (call-by-value default)
// Fig. 6.12: fig06_12.cpp
// Demonstrating that class objects can be assigned
// to each other using default memberwise copy
#include <iostream>

using std::cout;
using std::endl;

// Simple Date class
class Date {
public:
    Date( int = 1, int = 1, int = 1990 ); // default constructor
    void print();

private:
    int month;
    int day;
    int year;
};

// Simple Date constructor with no range checking
Date::Date( int m, int d, int y )
{  
    month = m;
    day = d;
    year = y;
}

// Print the Date in the form mm-dd-yyyy
void Date::print()
{  
    cout << month << '-' << day << '-' << year;  
}
```cpp
int main()
{
    Date date1( 7, 4, 1993 ), date2; // d2 defaults to 1/1/90
    cout << "date1 = ";
date1.print();
cout << "\ndate2 = ";
date2.print();

date2 = date1;   // assignment by default memberwise copy
    cout << "\n\nAfter default memberwise copy, date2 = ";
date2.print();
cout << endl;

    return 0;
}
```

Program Output:

- `date1` = 7-4-1993
- `date2` = 1-1-1990

After default memberwise copy, `date2` = 7-4-1993
6.17 Software Reusability

• Software resusability
  – Implementation of useful classes
  – Class libraries exist to promote reusability
    • Allows for construction of programs from existing, well-defined, carefully tested, well-documented, portable, widely available components
  – Speeds development of powerful, high-quality software