IS 0020
Program Design and Software Tools
Introduction to C++ Programming

Lecture 3

Jan 20, 2004
Quiz 1

• Average: about 3.8
• More than half obtained: 4 +
• Highest is 8
• Need more work/practice!
Quiz 1

• Question 1

```c
int i;
for ( i = 2; i < 20; i += i )
{
    printf("%d, ", i);
    i = i + 1;
}
```

Answer: 2, 6, 14
• Question 2

```c
int sum = 0; int i = 1, j = 1;
while (i = 5){
    sum = sum + i;
    if (sum == 20) break;
    j++;
}
printf( "%d, %d", i, j);
```

Answer: 5, 4
Quiz 1

• Question 3

```java
x = 0; y = 5;
do {
    switch (x) {
    case "2":
        y += 3 * x;
        break;
    case "3":
        y += x;
        break;
    default:
        y += x;
    }
} while (++x < 5);

Answer: x = 5, y = 25
```
Quiz 1

• Question 3

Consider the condition shown below. For what values of x and y will the values of z be as follows (state your answer in the back of the paper):

1. $z = 10$
2. $z = 20$

Condition: $((x == 5 || 7 < y && (100 - 2*50))?z = 10: z = 20)$

Answer 1: for $x = 5$ and any value of $y$
Answer 2: for $x \neq 5$ and any value of $y$
Function Pointers

• Arrays of pointers to functions
  – Menu-driven systems
  – Pointers to each function stored in array of pointers to functions
    • All functions must have same return type and same parameter types
  – Menu choice ➔ subscript into array of function pointers
// Fig. 5.26: fig05_26.cpp
// Demonstrating an array of pointers to functions.
#include <iostream>

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

// function prototypes
void function1( int );
void function2( int );
void function3( int );

int main()
{
   // initialize array of 3 pointers to functions that each
   // take an int argument and return void
   void (*f[ 3 ])( int ) = { function1, function2, function3 };

   int choice;

   cout << "Enter a number between 0 and 2, 3 to end: ";
   cin >> choice;
}
// process user's choice
while ( choice >= 0 && choice < 3 ) {
    // invoke function at location choice in array f
    // and pass choice as an argument
    (*f[ choice ])( choice );

    cout << "Enter a number between 0 and 2, 3 to end: ";
    cin >> choice;
}

cout << "Program execution completed." << endl;
return 0;  // indicates successful termination

} // end main

void function1( int a )
{
    cout << "You entered " << a
        << " so function1 was called\n\n";
}

} // end function1
void function2( int b )
{
    cout << "You entered " << b << " so function2 was called\n\n";
}

void function3( int c )
{
    cout << "You entered " << c << " so function3 was called\n\n";
}

Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called

Enter a number between 0 and 2, 3 to end: 3
Program execution completed.
Fundamentals of Characters and Strings

• Character constant
  – Integer value represented as character in single quotes
  – 'z' is integer value of z
    • 122 in ASCII

• String
  – Series of characters treated as single unit
  – Can include letters, digits, special characters +, -, *, ...
  – String literal (string constants)
    • Enclosed in double quotes, for example:
      "I like C++"
  – Array of characters, ends with null character ' \0 '
  – String is constant pointer
    • Pointer to string’s first character
      – Like arrays
Fundamentals of Characters and Strings

• String assignment
  – Character array
    • `char color[] = "blue";`
    – Creates 5 element `char` array `color`
      • last element is `\0`
  – Variable of type `char *`
    • `char *colorPtr = "blue";`
    – Creates pointer `colorPtr` to letter `b` in string “blue”
      • “blue” somewhere in memory
  – Alternative for character array
    • `char color[] = { ‘b’, ‘l’, ‘u’, ‘e’, ‘\0’ };`
Fundamentals of Characters and Strings

Reading strings

- Assign input to character array `word[20]`

```cpp
    cin >> word
```

- Reads characters until whitespace or EOF
- String could exceed array size

```cpp
    cin >> setw(20) >> word;
```

- Reads 19 characters (space reserved for \0)
Fundamentals of Characters and Strings

• **cin.getline**
  - Read line of text
  - `cin.getline( array, size, delimiter );`
  - Copies input into specified `array` until either
    • One less than `size` is reached
    • `delimiter` character is input
  - Example
    ```cpp
    char sentence[ 80 ];
    cin.getline( sentence, 80, '\n' );
    ```

• **cin.get()**
  - Read character and returns that character
  - Example
    ```cpp
    char c;
    c = cin.get();
    ```
    Could use a Condition like
    ```cpp
    ((c = cin.get())!= '\n');
    ```
String Manipulation Functions of the String-handling Library

String handling library `<cstring>` provides functions to

- Manipulate string data
- Compare strings
- Search strings for characters and other strings
- Tokenize strings (separate strings into logical pieces)
Classes
Introduction

- **Object-oriented programming (OOP)**
  - Encapsulates data (attributes) and functions (behavior) into packages called classes

- **Information hiding**
  - Class objects communicate across well-defined interfaces
  - Implementation details hidden within classes themselves

- **User-defined (programmer-defined) types: classes**
  - Data (data members)
  - Functions (member functions or methods)
  - Similar to blueprints – reusable
  - Class instance: object
Structure Definitions

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

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

- **Structure member naming**
  - In same `struct`: must have unique names
  - In different `struct`s: can share name

- **struct** definition must end with semicolon
Structure Definitions

• Self-referential structure
  – Structure member cannot be instance of enclosing `struct`
  – Structure member can be pointer to instance of enclosing `struct` (self-referential structure)
    • Used for linked lists, queues, stacks and trees

• `struct` definition
  – Creates new data type used to declare variables
  – Structure variables declared like variables of other types
  – Examples:
    • `Time timeObject;`
    • `Time timeArray[ 10 ];`
    • `Time *timePtr;`
    • `Time &timeRef = timeObject;`
Accessing Structure Members

• Member access operators
  – Dot operator ( . ) for structure and class members
  – Arrow operator ( \rightarrow ) for structure and class members via pointer to object
  – Print member \texttt{hour} of \texttt{timeObject}:
    \begin{verbatim}
    cout << timeObject.hour;
    \end{verbatim}
    OR
    \begin{verbatim}
    timePtr = &timeObject;
    cout << timePtr->hour;
    \end{verbatim}
  – \texttt{timePtr->hour} same as ( *\texttt{timePtr} ).\texttt{hour}

• Parentheses required
  – * lower precedence than .
Implementing a User-Defined Type Time with a struct

- Default: structures passed by value
  - Pass structure by reference
    - Avoid overhead of copying structure

- C-style structures
  - No “interface”
    - If implementation changes, all programs using that struct must change accordingly
  - Cannot print as unit
    - Must print/format member by member
  - Cannot compare in entirety
    - Must compare member by member
// Fig. 6.1: fig06_01.cpp
// Create a structure, set its members, and print it.
#include <iostream>

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

#include <iomanip>
using std::setfill;
using std::setw;

// structure definition
struct Time {
    int hour;     // 0-23 (24-hour clock format)
    int minute;   // 0-59
    int second;   // 0-59
}; // end struct Time

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

Define structure type **Time** with three integer members.

Pass references to constant **Time** objects to eliminate copying overhead.
int main()
{
    Time dinnerTime;         // variable of new type Time

    dinnerTime.hour = 18;    // set hour member of dinnerTime
    dinnerTime.minute = 30;  // set minute member of dinnerTime
    dinnerTime.second = 0;   // set second member of dinnerTime

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

    dinnerTime.hour = 29;    // set hour to invalid value
    dinnerTime.minute = 73;  // set minute to invalid value

    cout << "\nTime with invalid values: ";
    printUniversal( dinnerTime );
    cout << endl;

    return 0;
}

// print time in universal-time format
void printUniversal( const Time &t )
{
    cout << setfill( '0' ) << setw( 2 ) << t.hour << " : "
    << setw( 2 ) << t.minute << " : "
    << setw( 2 ) << t.second;
}

// end function printUniversal

// print time in standard-time format
void printStandard( const Time &t )
{
    cout << ( ( t.hour == 0 || t.hour == 12 ) ?
    12 : t.hour % 12 ) << ":" << setfill( '0' )
    << setw( 2 ) << t.minute << " : "
    << setw( 2 ) << t.second
    << ( t.hour < 12 ? " AM" : " PM" );
}

} // end function printStandard

Dinner will be held at 18:30:00 universal time, which is 6:30:00 PM standard time. Time with invalid values: 29:73:00

Use parameterized stream manipulator setfill.

Use dot operator to access data members.
Implementing a Time Abstract Data Type with a class

• Classes
  – Model objects
    • Attributes (data members)
    • Behaviors (member functions)
  – Defined using keyword `class`
  – Member functions
    • Methods
      • Invoked in response to messages

• Member access specifiers
  – `public:`
    • Accessible wherever object of class in scope
  – `private:`
    • Accessible only to member functions of class
  – `protected:`
Implementing a Time Abstract Data Type with a class

Constructor function
- Special member function
  - Initializes data members
  - Same name as class
- Called when object instantiated
- Several constructors
  - Function overloading
- No return type
```cpp
class Time {
    public:
        Time();                       // constructor
        void setTime(int, int, int); // set hour, minute, second
        void printUniversal();        // print universal-time format
        void printStandard();          // print standard-time format
    private:
        int hour;     // 0 - 23 (24-hour clock format)
        int minute;  // 0 - 59
        int second;   // 0 - 59
    }; // end class Time
```
Implementing a `Time` Abstract Data Type with a class

- **Objects of class**
  - After class definition
    - Class name new type specifier
      - C++ extensible language
    - Object, array, pointer and reference declarations
  - Example:

```
Time sunset;                   // object of type Time
Time arrayOfTimes[ 5 ];       // array of Time objects
Time *pointerToTime;          // pointer to a Time object
Time &dinnerTime = sunset;    // reference to a Time object
```
Implementing a Time Abstract Data Type with a class

- Member functions defined outside class
  - Binary scope resolution operator (::)
    - “Ties” member name to class name
    - Uniquely identify functions of particular class
    - Different classes can have member functions with same name
  - Format for defining member functions
    
    ```
    ReturnType ClassName::MemberFunctionName() {
        ...
    }
    ```
    - Does not change whether function public or private

- Member functions defined inside class
  - Do not need scope resolution operator, class name
  - Compiler attempts inline
    - Outside class, inline explicitly with keyword inline
// Fig. 6.3: fig06_03.cpp
// Time class.
#include <iostream>

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

#include <iomanip>

using std::setfill;
using std::setw;

// Time abstract data type (ADT) definition
class Time {

public:
    Time();                       // constructor
    void setTime( int, int, int ); // set hour, minute, second
    void printUniversal();        // print universal-time format
    void printStandard();          // print standard-time format

};
private:
  int hour;    // 0 - 23 (24-hour clock format)
  int minute;  // 0 - 59
  int second;  // 0 - 59

}; // end class Time

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

} // end Time constructor

// set new Time value using universal time, perform validity
// checks on the data values and 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;
}

} // end function setTime

Constructor initializes private data members to 0.

Public member function checks parameter values for validity before setting private data members.
47 // print Time in universal format
48 void Time::printUniversal()
49 {
50     cout << setfill( '0' ) << setw( 2 ) << hour << " : "
51         << setw( 2 ) << minute << " : "
52         << setw( 2 ) << second;
53 }
54 } // end function printUniversal
55
56 // print Time in standard format
57 void Time::printStandard()
58 {
59     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
60         << " : " << setfill( '0' ) << setw( 2 ) << minute
61         << " : " << setw( 2 ) << second
62         << ( hour < 12 ? " AM" : " PM" );
63 }
64 } // end function print
65
66 int main()
67 {
68     Time t;  // instantiate object t of class Time
// output Time object t's initial values
cout << "The initial universal time is ";
t.printUniversal(); // 00:00:00

cout << "\nThe initial standard time is ";
t.printStandard(); // 12:00:00 AM

t.setTime( 13, 27, 6 ); // change time

// output Time object t's new values
cout << "\nUniversal time after setTime is ";
t.printUniversal(); // 13:27:06

cout << "\nStandard time after setTime is ";
t.printStandard(); // 1:27:06 PM

t.setTime( 99, 99, 99 ); // attempt invalid settings

// output t's values after specifying invalid values
cout << "\n\nAfter attempting invalid settings:
Universal time: ";
t.printUniversal(); // 00:00:00
cout << "\nStandard time: ";
t.printStandard();    // 12:00:00 AM
cout << endl;
return 0;
} // end main

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

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

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM

Data members set to 0 after attempting invalid settings.
Implementing a Time Abstract Data Type with a class

• Destructors
  – Same name as class
    • Preceded with tilde (\texttt{\textasciitilde})
  – No arguments
  – Cannot be overloaded
  – Performs “termination housekeeping”
Implementing a *Time* Abstract Data Type with a class

- Advantages of using classes
  - Simplify programming
  - Interfaces
    - Hide implementation
  - Software reuse
    - Composition (aggregation)
      - Class objects included as members of other classes
    - Inheritance
      - New classes derived from old
Class Scope and Accessing Class Members

- **Class scope**
  - Data members, member functions
  - Within class scope
    - Class members
      - Immediately accessible by all member functions
      - Referenced by name
    - Outside class scope
      - Referenced through handles
        - Object name, reference to object, pointer to object
  
- **File scope**
  - Nonmember functions
Class Scope and Accessing Class Members

- Function scope
  - Variables declared in member function
  - Only known to function
  - Variables with same name as class-scope variables
    - Class-scope variable “hidden”
      - Access with scope resolution operator (::)
        \texttt{ClassName::classVariableName}
  - Variables only known to function they are defined in
  - Variables are destroyed after function completion
Class Scope and Accessing Class Members

- **Operators to access class members**
  - Identical to those for `struct`
  - Dot member selection operator ( . )
    - Object
    - Reference to object
  - Arrow member selection operator ( → )
    - Pointers
// 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;

// class Count definition
class Count {
public:
    int x;

    void print()
    {
        cout << x << endl;
    }
}; // end class Count

Data member x public to illustrate class member access operators; typically data members private.
```cpp
int main()
{
    Count counter;                // create counter object
    Count *counterPtr = &counter; // create pointer to counter
    Count &counterRef = counter;  // create reference to counter

    cout << "Assign 1 to x and print using the object's name: " << counter.x = 1; // assign
    counter.print();              // call member function print

    cout << "Assign 2 to x and print using a reference: " << counterRef.x = 2; // assign
    counterRef.print();           // call member function print

    cout << "Assign 3 to x and print using a pointer: " << counterPtr->x = 3; // assign
    counterPtr->print();          // call member function print

    return 0;
}
```

Assign 1 to x and print using the object's name: 1
Assign 2 to x and print using a reference: 2
Assign 3 to x and print using a pointer: 3

Use dot member selection operator for `counter` object.
Use dot member selection operator for `counterRef` reference to object.
Use arrow member selection operator for `counterPtr` pointer to object.
Separating Interface from Implementation

• Separating interface from implementation
  – Advantage
    • Easier to modify programs
  – Disadvantage
    • Header files
      – Portions of implementation
        • Inline member functions
      – Hints about other implementation
        • private members
    • Can hide more with proxy class
Separating Interface from Implementation

• Header files
  – Class definitions and function prototypes
  – Included in each file using class
    • `#include`
  – File extension `.h`

• Source-code files
  – Member function definitions
  – Same base name
    • Convention
  – Compiled and linked
// Fig. 6.5: time1.h
// Declaration of class Time.
// Member functions are defined in
// prevent multiple inclusions of header file
#elseif 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 printUniversal();         // print universal-time format
    void printStandard();          // print standard-time format

private:
    int hour;     // 0 - 23 (24-hour clock format)
    int minute;   // 0 - 59
    int second;   // 0 - 59

}; // end class Time

#endif

Preprocessor code to prevent multiple inclusions.

Code between these directives  "If not defined"  Preprocessor directive defines
Naming convention:  header file name with underscore replacing period.
// Fig. 6.6: time1.cpp
// Member-function definitions for class Time.
#include <iostream>

using std::cout;

#include <iomanip>

using std::setw;

#include "time1.h"

// include definition of class Time from time1.h

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

// end Time constructor
```cpp
23  // Set new Time value using universal time. Perform validity
24  // checks on the data values. Set invalid values to zero.
25  void Time::setTime( int h, int m, int s )
26  {
27      hour = ( h >= 0 && h < 24 ) ? h : 0;
28      minute = ( m >= 0 && m < 60 ) ? m : 0;
29      second = ( s >= 0 && s < 60 ) ? s : 0;
30  }
31  } // end function setTime
32
33  // print Time in universal format
34  void Time::printUniversal()
35  {
36      cout << setfill( '0' ) << setw( 2 ) << hour << ":";
37      << setw( 2 ) << minute << ":";
38      << setw( 2 ) << second;
39  }
40  } // end function printUniversal
41```
// print Time in standard format
void Time::printStandard()
{
    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
         << " :" << setfill( '0' ) << setw( 2 ) << minute
         << " :" << setw( 2 ) << second
         << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
// Fig. 6.7: fig06_07.cpp
// Program to test class Time.
// NOTE: This file must be compiled with time1.cpp.
#include <iostream>

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

#include "time1.h"

int main()
{
    Time t; // instantiate object t of class Time

    // output Time object t's initial values
    cout << "The initial universal time is ";
t.printUniversal(); // 00:00:00
    cout << "\nThe initial standard time is ";
t.printStandard(); // 12:00:00 AM

    t.setTime( 13, 27, 6 ); // change time
// output Time object t's new values
cout << "\nUniversal time after setTime is ";
t.printUniversal(); // 13:27:06
cout << "\nStandard time after setTime is ";
t.printStandard(); // 1:27:06 PM

t.setTime( 99, 99, 99 ); // attempt invalid settings

// output t's values after specifying invalid values
cout << "\nAfter attempting invalid settings:" << "\nUniversal time: ";
t.printUniversal(); // 00:00:00
cout << "\nStandard time: ";
t.printStandard(); // 12:00:00 AM
cout << endl;

return 0;

} // end main

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

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM
Controlling Access to Members

• Access modes
  - `private`
    • Default access mode
    • Accessible to member functions and `friends`
  - `public`
    • Accessible to any function in program with handle to class object
  - `protected`
    • later
// Fig. 6.8: fig06_08.cpp
// Demonstrate errors resulting from attempts
// to access private class members.
#include <iostream>

using std::cout;

// include definition of class Time from timel.h
#include "timel.h"

int main()
{
    Time t; // create Time object
    t.hour = 7; // error: 'Time::hour' is not accessible
    // error: 'Time::minute' is not accessible
    cout << "minute = " << t.minute;
    return 0;
} // end main
Controlling Access to Members

• Class member access
  – Default \textit{private}
  – Explicitly set to \textit{private, public, protected}

• \texttt{struct} member access
  – Default \textit{public}
  – Explicitly set to \textit{private, public, protected}

• Access to class’s \textit{private} data
  – Controlled with access functions (accessor methods)
    • Get function
      – Read \textit{private} data
    • Set function
      – Modify \textit{private} data
Access Functions and Utility Functions

- **Access functions**
  - *public*
  - Read/display data
  - Predicate functions
    - Check conditions

- **Utility functions (helper functions)**
  - *private*
  - Support operation of *public* member functions
  - Not intended for direct client use
// Fig. 6.9: salesp.h
// SalesPerson class definition.
// Member functions defined in salesp.cpp.
#ifndef SALESP_H
#define SALESP_H

class SalesPerson {
public:
    SalesPerson();                // constructor
    void getSalesFromUser();      // input sales from keyboard
    void setSales( int, double ); // set sales for a month
    void printAnnualSales();      // summarize and print sales

private:
    double totalAnnualSales(); // utility function
    double sales[ 12 ];        // 12 monthly sales figures

}; // end class SalesPerson

#endif
SalesPerson::SalesPerson()
{
    for ( int i = 0; i < 12; i++ )
        sales[ i ] = 0.0;
} // end SalesPerson constructor
25 // get 12 sales figures from the user at the keyboard
26 void SalesPerson::getSalesFromUser()
27 {
28     double salesFigure;
29
30     for ( int i = 1; i <= 12; i++ ) {
31         cout << "Enter sales amount for month " << i << ": ";
32         cin >> salesFigure;
33         setSales( i, salesFigure );
34     }
35 } // end for
36
37 } // end function getSalesFromUser
38
39 // set one of the 12 monthly sales figures; function subtracts
40 // one from month value for proper subscript.
41 void SalesPerson::setSales( int month, double amount )
42 {
43     // test for valid month and amount values
44     if ( month >= 1 && month <= 12 && amount > 0 )
45         sales[ month - 1 ] = amount; // adjust for subscripts 0-11
46     else // invalid month or amount value
47         cout << "Invalid month or sales figure" << endl;
48 } // end function setSales
49
Set access function performs validity checks.
49
50 ) // end function setSales
51
52 // print total annual sales (with help of utility function)
53 void SalesPerson::printAnnualSales()
54 {
55     cout << setprecision( 2 ) << fixed
56       << "\nThe total annual sales are: $" << totalAnnualSales() << endl; // call utility function
57
58 } // end function printAnnualSales
59
60 // private utility function to total annual sales
61 double SalesPerson::totalAnnualSales()
62 {
63     double total = 0.0; // initialize total
64
65     for ( int i = 0; i < 12; i++ ) // summarize sales results
66         total += sales[ i ];
67
68     return total;
69
70 } // end function totalAnnualSales

private utility function to help function
printAnnualSales;
encapsulates logic of
manipulating sales array.
// Fig. 6.11: fig06_11.cpp
// Demonstrating a utility function.
// Compile this program with salesp.cpp

// include SalesPerson class definition from salesp.h
#include "salesp.h"

int main()
{
    SalesPerson s; // create SalesPerson object
    s.getSalesFromUser(); // note simple sequential code
    s.printAnnualSales(); // control structures in main
    return 0;
} // end main

Simple sequence of member function calls; logic encapsulated in member functions.
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
Initializing Class Objects: Constructors

- **Constructors**
  - Initialize data members
    - Or can set later
  - Same name as class
  - No return type

- **Initializers**
  - Passed as arguments to constructor
  - In parentheses to right of class name before semicolon

```java
Class-type ObjectName( value1, value2, ...);
```
Using Default Arguments with Constructors

- **Constructors**
  - Can specify default arguments
  - Default constructors
    - Defaults all arguments
      - OR
    - Explicitly requires no arguments
    - Can be invoked with no arguments
    - Only one per class
// Fig. 6.12: time2.h
// Declaration of class Time.
// Member functions defined in time2.cpp.

// 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 printUniversal();         // print universal-time format
    void printStandard();          // print standard-time format

  private:
    int hour;     // 0 - 23 (24-hour clock format)
    int minute;   // 0 - 59
    int second;  // 0 - 59

}; // end class Time

#endif
// Fig. 6.13: time2.cpp
// Member-function definitions for class Time.
#include <iostream>

using std::cout;

#include <iomanip>

using std::setfill;

using std::setw;

// include definition of class Time from time2.h
#include "time2.h"

// Time constructor initializes each data member to zero;
// ensures all Time objects start in a consistent state
Time::Time( int hr, int min, int sec )
{
    setTime( hr, min, sec ); // validate and set time
} // end Time constructor
// set new Time value using universal time, perform validity
// checks on the data values and 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 universal format
void Time::printUniversal()
{
    cout << setfill( '0' ) << setw( 2 ) << hour << "::" << setfill( '0' ) << setw( 2 ) << minute << "::" << ( 2
// Fig. 6.14: fig06_14.cpp
// Demonstrating a default constructor for class Time.
#include <iostream>

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

// include definition of class Time from time2.h
#include "time2.h"

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

    cout << "Constructed with:

    " all default arguments: \n    ";
t1.printUniversal(); // 00:00:00
    cout << "\n    ";
t1.printStandard(); // 12:00:00 AM
}
cout << "\n\nhour specified; default minute and second:\n \n";
t2.printUniversal();  // 02:00:00
cout << "\n ";
t2.printStandard();    // 2:00:00 AM

cout << "\n\nhour and minute specified; default second:\n \n";
t3.printUniversal();  // 21:34:00
cout << "\n ";
t3.printStandard();    // 9:34:00 PM

cout << "\n\nhour, minute, and second specified:\n \n";
t4.printUniversal();  // 12:25:42
cout << "\n ";
t4.printStandard();    // 12:25:42 PM

cout << "\n\nall invalid values specified:\n \n";
t5.printUniversal();  // 00:00:00
cout << "\n ";
t5.printStandard();    // 12:00:00 AM
cout << endl;

return 0;

}  // end main
Destructors

- Destructors
  - Special member function
  - Same name as class
    - Preceded with tilde (~)
  - No arguments
  - No return value
  - Cannot be overloaded
  - Performs “termination housekeeping”
    - Before system reclaims object’s memory
      - Reuse memory for new objects
  - No explicit destructor
    - Compiler creates “empty” destructor”
When Constructors and Destructors Are Called

• Constructors and destructors
  – Called implicitly by compiler

• Order of function calls
  – Depends on order of execution
    • When execution enters and exits scope of objects
  – Generally, destructor calls reverse order of constructor calls
When Constructors and Destructors Are Called

- **Order of constructor, destructor function calls**
  - Global scope objects
    - Constructors
      - Before any other function (including `main`)
    - Destructors
      - When `main` terminates (or `exit` function called)
      - Not called if program terminates with `abort`
  - Automatic local objects
    - Constructors
      - When objects defined
        - Each time execution enters scope
    - Destructors
      - When objects leave scope
        - Execution exits block in which object defined
      - Not called if program ends with `exit` or `abort`
When Constructors and Destructors Are Called

• Order of constructor, destructor function calls
  - **static** local objects
    - Constructors
      - Exactly once
      - When execution reaches point where object defined
    - Destructors
      - When `main` terminates or `exit` function called
      - Not called if program ends with `abort`
// Fig. 6.15: create.h
// Definition of class CreateAndDestroy.
// Member functions defined in create.cpp.
#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy {

public:

    CreateAndDestroy( int, char * ); // constructor

    ~CreateAndDestroy();

private:

    int objectID;

    char *message;

}; // end class CreateAndDestroy

#endif

Constructor and destructor member functions.

private members to show order of constructor,
destructor function calls.
// Fig. 6.16: create.cpp  
// Member-function definitions for class CreateAndDestroy  
#include <iostream>  

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

// include CreateAndDestroy class definition from create.h  
#include "create.h"  

// constructor  
CreateAndDestroy::CreateAndDestroy(  
    int objectNumber, char *messagePtr )  
{  
    objectId = objectNumber;  
    message = messagePtr;  
    cout << "Object " << objectId << " constructor runs "  
        << message << endl;  
} // end CreateAndDestroy constructor  

Output message to  
demonstrate timing of  
constructor function calls.
// destructor
CreateAndDestroy::~CreateAndDestroy()
{
    // the following line is for pedagogic purposes only
    cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );

    cout << "Object " << objectID << " destructor runs "
         << message << endl;
}

// end ~CreateAndDestroy destructor
// Fig. 6.17: fig06_17.cpp
// Demonstrating the order in which constructors and
// destructors are called.
#include <iostream>

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

// include CreateAndDestroy class definition from create.h
#include "create.h"

void create( void ); // prototype

// global object
CreateAndDestroy first( 1, "(global before main)" );

int main()
{
    cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
    CreateAndDestroy second( 2, "(local automatic in main)" );
    static CreateAndDestroy third( 3, "(local static in main)" );
}
create(); // call function to create objects

cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
CreateAndDestroy fourth( 4, "(local automatic in main)" );
cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
return 0;
}

// function to create objects
void create( void )
{
    cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
CreateAndDestroy fifth( 5, "(local automatic in create)" );
static CreateAndDestroy sixth( 6, "(local static in create)" );
CreateAndDestroy seventh( 7, "(local automatic in create)" );

} // end main
cout << "\nCREATE FUNCTION: EXECUTION ENDS\" << endl;
} // end function create
Object 1 constructor runs (global before main)

MAIN FUNCTION: EXECUTION BEGINS
Object 2 constructor runs (local automatic in main)
Object 3 constructor runs (local static in main)

CREATE FUNCTION: EXECUTION BEGINS
Object 5 constructor runs (local automatic in create)
Object 6 constructor runs (local static in create)
Object 7 constructor runs (local automatic in create)

CREATE FUNCTION: EXECUTION ENDS
Object 7 destructor runs (local automatic in create)
Object 5 destructor runs (local automatic in create)

MAIN FUNCTION: EXECUTION RESUMES
Object 4 constructor runs (local automatic in main)

MAIN FUNCTION: EXECUTION ENDS
Object 4 destructor runs (local automatic in main)
Object 2 destructor runs (local automatic in main)
Object 6 destructor runs (local static in create)
Object 3 destructor runs (local static in main)
Object 1 destructor runs (global before main)
Using Set and Get Functions

• Set functions
  – Perform validity checks before modifying private data
  – Notify if invalid values
  – Indicate with return values

• Get functions
  – “Query” functions
  – Control format of data returned
// Fig. 6.18: time3.h
// Declaration of class Time.
// Member functions defined in time3.cpp

// prevent multiple inclusions of header file
#ifndef TIME3_H
#define TIME3_H

class Time {

public:
    Time( int = 0, int = 0, int = 0 );  // default constructor

    // set functions
    void setTime( int, int, int );  // set hour, minute, second
    void setHour( int );   // set hour
    void setMinute( int ); // set minute
    void setSecond( int ); // set second

    // get functions
    int getHour();         // return hour
    int getMinute();      // return minute
    int getSecond();       // return second

}; // end class Time
#endif // TIME3_H

Set functions.
Get functions.
void printUniversal(); // output universal-time format
void printStandard();  // output standard-time format

private:
int hour;              // 0 - 23 (24-hour clock format)
int minute;            // 0 - 59
int second;            // 0 - 59

}; // end class Time

#endif
Fig. 6.19: time3.cpp

// Member-function definitions for Time class.

#include <iostream>

using std::cout;

#include <iomanip>

using std::setfill;

using std::setw;

// include definition of class Time from time3.h
#include "time3.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 );
}

// end Time constructor
// set hour, minute and second values
void Time::setTime( int h, int m, int s )
{
    setHour( h );
    setMinute( m );
    setSecond( s );
}

// set hour value
void Time::setHour( int h )
{
    hour = ( h >= 0 && h < 24 )? h : 0;
}

// set minute value
void Time::setMinute( int m )
{
    minute = ( m >= 0 && m < 60 )? m : 0;
}
Set function performs validity checks before modifying data.

Get functions allow client to read data.
Get function allows client to read data.
// Fig. 6.20: fig06_20.cpp
// Demonstrating the Time class set and get functions
#include <iostream>

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

// include definition of class Time from time3.h
#include "time3.h"

void incrementMinutes( Time &, const int );  // prototype

int main()
{
    Time t;              // create Time object
    // set time using individual set functions
    t.setHour( 17 );     // set hour to valid value
    t.setMinute( 34 );   // set minute to valid value
    t.setSecond( 25 );   // set second to valid value
}
// use get functions to obtain hour, minute and second
cout << "Result of setting all valid values: \n"
  << "  Hour: " << t.getHour() 
  << "  Minute: " << t.getMinute() 
  << "  Second: " << t.getSecond();

// set time using individual set functions
  t.setHour( 234 );    // invalid hour set to 0 
  t.setMinute( 43 );   // set minute to valid value 
  t.setSecond( 6373 ); // invalid second set to 0 

// display hour, minute and second after setting 
// invalid hour and second values 
  cout << "\n\nResult of attempting to set invalid hour and"
  << "  second: \n  Hour: " << t.getHour() 
  << "  Minute: " << t.getMinute() 
  << "  Second: " << t.getSecond() << "\n\n";

  t.setTime( 11, 58, 0 );    // set time 
  incrementMinutes( t, 3 );  // increment t's minute by 3

return 0;

} // end main
// add specified number of minutes to a Time object
void incrementMinutes( Time &tt, const int count )
{
    cout << "Incrementing minute " << count
         << " times:\nStart time: ";
    tt.printStandard();

    for ( int i = 0; i < count; i++ ) {
        tt.setMinute( ( tt.getMinute() + 1 ) % 60 );

        if ( tt.getMinute() == 0 )
            tt.setHour( ( tt.getHour() + 1 ) % 24);

        cout << " minute + 1: ";
        tt.printStandard();
    } // end for

    cout << endl;
} // end function incrementMinutes

Using get functions to read data and set functions to modify data.
Result of setting all valid values:
  Hour: 17  Minute: 34  Second: 25

Result of attempting to set invalid hour and second:
  Hour: 0  Minute: 43  Second: 0

Incrementing minute 3 times:
Start time: 11:58:00 AM
minute + 1: 11:59:00 AM
minute + 1: 12:00:00 PM
minute + 1: 12:01:00 PM

Attempting to set data members with invalid values results in error message and members set to 0.
Subtle Trap: Returning a Reference to a private Data Member

• Reference to object
  – \&pRef = p;
  – Alias for name of object
  – Lvalue
    • Can receive value in assignment statement
      – Changes original object

• Returning references
  – public member functions can return non-const references to private data members
    • Client able to modify private data members
// Fig. 6.21: time4.h
// Declaration of class Time.
// Member functions defined in time4.cpp

// prevent multiple inclusions of header file
#ifndef TIME4_H
#define TIME4_H

class Time {

public:
    Time( int = 0, int = 0, int = 0 );
    void setTime( int, int, int );
    int getHour();

    int &badSetHour( int );  // DANGEROUS reference return

private:
    int hour;
    int minute;
    int second;

}; // end class Time

#endif
// Fig. 6.22: time4.cpp
// Member-function definitions for Time class.

// include definition of class Time from time4.h
#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 );
}

} // end Time constructor

// set 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;

} // end function setTime
// return 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
}

// end function badSetHour
// Fig. 6.23: fig06_23.cpp
// Demonstrating a public member function that
// returns a reference to a private data member.
#include <iostream>

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

// include definition of class Time from time4.h
#include "time4.h"

int main()
{
    Time t;
    // store in hourRef the reference returned by badSetHour
    int &hourRef = t.badSetHour( 20 );
    cout << "Hour before modification: " << hourRef;
    // use hourRef to set invalid value in Time object t
    hourRef = 30;
    cout << "\nHour after modification: " << t.getHour();
}
// Dangerous: Function call that returns
// a reference can be used as an lvalue!

t.badSetHour( 12 ) = 74;

cout << "\n\n*******************************\n"
  << "POOR PROGRAMMING PRACTICE!!!!!!\n"
  << "badSetHour as an lvalue, Hour: \n"
  << t.getHour() 
  << "\n*******************************" << endl;

return 0;

} // end main
Default Memberwise Assignment

• Assigning objects
  – Assignment operator (=)
    • Can assign one object to another of same type
    • Default: memberwise assignment
      – Each right member assigned individually to left member

• Passing, returning objects
  – Objects passed as function arguments
  – Objects returned from functions
  – Default: pass-by-value
    • Copy of object passed, returned
      – Copy constructor
        • Copy original values into new object
// Fig. 6.24: fig06_24.cpp
// Demonstrating that class objects can be assigned
// to each other using default memberwise assignment.
#include <iostream>

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

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

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

// end Date constructor

// print Date in the format mm-dd-yyyy
void Date::print()
{
    cout << month << '-' << day << '-' << year;
}

// end function print

int main()
{
    Date date1( 7, 4, 2002 );
    Date date2; // date2 defaults to 1/1/1990
default = 7-4-2002
date2 = 1-1-1990

After default memberwise assignment, date2 = 7-4-2002
Software Reusability

• Software reusability
  – Class libraries
    • Well-defined
    • Carefully tested
    • Well-documented
    • Portable
    • Widely available
  – Speeds development of powerful, high-quality software
    • Rapid applications development (RAD)
  – Resulting problems
    • Cataloging schemes
    • Licensing schemes
    • Protection mechanisms
**const (Constant) Objects and const Member Functions**

- **Principle of least privilege**
  - Only allow modification of necessary objects

- **Keyword `const`**
  - Specify object not modifiable
  - Compiler error if attempt to modify `const` object
  - Example
    ```
    const Time noon( 12, 0, 0 );
    ```
  - Declares `const` object `noon` of class `Time`
  - Initializes to 12
**const (Constant) Objects and const Member Functions**

**const member functions**

- Member functions for **const** objects must also be **const**
  - Cannot modify object
- Specify **const** in both prototype and definition
  - Prototype
    - After parameter list
  - Definition
    - Before beginning left brace
const (Constant) Objects and const Member Functions

• Constructors and destructors
  – Cannot be const
  – Must be able to modify objects
    • Constructor
      – Initializes objects
    • Destructor
      – Performs termination housekeeping
// Fig. 7.1: time5.h
// Definition of class Time.
// Member functions defined in time5.cpp.
#ifndef TIME5_H
#define TIME5_H

class Time {
    public:
    Time( int = 0, int = 0, int = 0 ); // default constructor

    // set functions
    void setTime( int, int, int ); // set time
    void setHour( int );           // set hour
    void setMinute( int );         // set minute
    void setSecond( int );         // set second

    // get functions (normally declared const)
    int getHour() const;           // return hour
    int getMinute() const;         // return minute
    int getSecond() const;         // return second

    // print functions (normally declared const)
    void printUniversal() const;   // print universal time
    void printStandard();          // print standard time
}
#define TIME5_H

Declare const get functions.

Declare const function printUniversal.
private:
  int hour;    // 0 - 23 (24-hour clock format)
  int minute;  // 0 - 59
  int second;  // 0 - 59
}; // end class Time

#ifdef
// Fig. 7.2: time5.cpp
// Member-function definitions for class Time.
#include <iostream>

using std::cout;

#include <iomanip>
using std::setfill;
using std::setw;

// include definition of class Time from time5.h
#include "time5.h"

// constructor function to initialize private data;
// calls member function setTime to set variables;
// default values are 0 (see class definition)
Time::Time( int hour, int minute, int second )
{
    setTime( hour, minute, second );
}

// end Time constructor
// set hour, minute and second values
void Time::setTime( int hour, int minute, int second )
{
    setHour( hour );
    setMinute( minute );
    setSecond( second );
}

// set hour value
void Time::setHour( int h )
{
    hour = ( h >= 0 && h < 24 ) ? h : 0;
}

// set minute value
void Time::setMinute( int m )
{
    minute = ( m >= 0 && m < 60 ) ? m : 0;
}
```cpp
47 // set second value
48 void Time::setSecond( int s )
49 {
50     second = ( s >= 0 && s < 60 ) ? s : 0;
51 }
52 } // end function setSecond
53
54 // return hour value
55 int Time::getHour() const
56 {
57     return hour;
58 }
59 } // end function getHour
60
61 // return minute value
62 int Time::getMinute() const
63 {
64     return minute;
65 }
66 } // end function getMinute
```

`const` functions do not modify objects.
// return second value
int Time::getSecond() const
{
    return second;
}

// print Time in universal format
void Time::printUniversal() const
{
    cout << setfill( '0' ) << setw( 2 ) << hour << "::"
        << setw( 2 ) << minute << "::"
        << setw( 2 ) << second;
}

// print Time in standard format
void Time::printStandard() // note lack of const declaration
{
    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << "::" << setfill( '0' ) << setw( 2 ) << minute
        << "::" << setw( 2 ) << second
        << ( hour < 12 ? " AM" : " PM" );
}

const functions do not modify objects.
// Fig. 7.3: fig07_03.cpp
// Attempting to access a const object with
// non-const member functions.

// include Time class definition from time5.h
#include "time5.h"

int main()
{
    Time wakeUp( 6, 45, 0 );      // non-constant object
    const Time noon( 12, 0, 0 );  // constant object

**const (Constant) Objects and const Member Functions**

- **Member initializer syntax**
  - Initializing with member initializer syntax
    - Can be used for
      - All data members
    - Must be used for
      - `const` data members
      - Data members that are references
// Fig. 7.4: fig07_04.cpp
// Using a member initializer to initialize a
// constant of a built-in data type.
#include <iostream>

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

class Increment {

public:

Increment( int c = 0, int i = 1 ); // default constructor

void addIncrement()
{
    count += increment;
}

} // end function addIncrement

void print() const; // prints count and increment
private:
  int count;
  const int increment;  // const data member
}; // end class Increment

// constructor
Increment::Increment( int c, int i ) : count( c ),  // initializer for non-const member
    increment( i )  // required initializer for const member
{
  // empty body
}

// print count and increment values
void Increment::print() const
{
  cout << "count = " << count
    << " , increment = " << increment << endl;
}

Declare increment as const
data member.
Member initializer list
increment as const
separated by colon.
Member initializer syntax can
be used for non-const
data member count.
Member initializer syntax
must be used for const data
member increment.
Member initializer consists of
data member name
(increment) followed by
parentheses containing initial
value (c).
```cpp
int main()
{
    Increment value(10, 5);

    cout << "Before incrementing: ";
    value.print();

    for (int j = 0; j < 3; j++) {
        value.addIncrement();
        cout << "After increment " << j + 1 << "": ";
        value.print();
    }

    return 0;
} // end main
```

Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
After increment 3: count = 25, increment = 5
Composition: Objects as Members of Classes

• Composition
  – Class has objects of other classes as members

• Construction of objects
  – Member objects constructed in order declared
    • Not in order of constructor’s member initializer list
    • Constructed before enclosing class objects (host objects)
// Fig. 7.6: date1.h
// Date class definition.
// Member functions defined in date1.cpp
#ifndef DATE1_H
#define DATE1_H

class Date {

public:
    Date( int = 1, int = 1, int = 1900 ); // default constructor
    void print() const; // print date in month/day/year format
    ~Date();  // provided to confirm destruction order

private:
    int month; // 1-12 (January-December)
    int day;   // 1-31 based on month
    int year;  // any year

    // utility function to test proper day for month and year
    int checkDay( int ) const;

}; // end class Date

#endif
// Fig. 7.7: date1.cpp
// Member-function definitions for class Date.
#include <iostream>

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

// include Date class definition from date1.h
#include "date1.h"

// constructor confirms proper value for month; calls
// utility function checkDay to confirm proper value for day
Date::Date( int mn, int dy, int yr )
{
    if ( mn > 0 && mn <= 12 ) // validate the month
        month = mn;
    else {                     // invalid month set to 1
        month = 1;
        cout << "Month " << mn << " invalid. Set to month 1.";
    }

    year = yr; // should validate yr
    day = checkDay( dy ); // validate the day
// Fig. 7.8: employee1.h
// Employee class definition.
// Member functions defined in employee1.cpp.
#ifndef EMPLOYEE1_H
#define EMPLOYEE1_H

// include Date class definition from date1.h
#include "date1.h"

class Employee {

public:
    Employee(const char *, const char *, const Date &, const Date &);

    void print() const;
    ~Employee();  // provided to confirm destruction order

private:
    char firstName[25];
    char lastName[25];
    const Date birthDate;  // composition: member object
    const Date hireDate;   // composition: member object

}; // end class Employee

// Using composition; Employee object contains Date objects as data members.
// Fig. 7.9: employee1.cpp
// Member-function definitions for class Employee.
#include <iostream>

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

#include <cstring>  // strcpy and strlen prototypes

#include "employee1.h"  // Employee class definition
#include "date1.h"      // Date class definition
13 // constructor uses member initializer list to pass initializer
14 // values to constructors of member objects birthDate and hireDate [Note: This invokes the so-called "default copy constructor"
15 // constructor" which the C++ compiler provides implicitly.]
16 Employee::Employee(const char *first, const char *last,
17 const Date &dateOfBirth, const Date &dateOfHire)
18   : birthDate(dateOfBirth), // initialize birthDate
19     hireDate(dateOfHire)     // initialize hireDate
20 {
21   // copy first into firstName and be sure that it fits
22   int length = strlen(first);
23   length = (length < 25 ? length : 24);
24   strncpy(firstName, first, length);
25   firstName[length] = '\0';
26
27   // copy last into lastName and be sure that it fits
28   length = strlen(last);
29   length = (length < 25 ? length : 24);
30   strncpy(lastName, last, length);
31   lastName[length] = '\0';
32
33   // output Employee object to show when constructor is called
34   cout << "Employee object constructor: "
35       << firstName << ' ' << lastName << endl;
38 } // end Employee constructor
39
40 // print Employee object
41 void Employee::print() const
42 {
43    cout << lastName << "", " << firstName << "\nHired: ";
44    hireDate.print();
45    cout << " Birth date: ";
46    birthDate.print();
47    cout << endl;
48
49 } // end function print
50
51 // output Employee object to show when it is deleted
52 Employee::~Employee()
53 {
54    cout << "Employee object destructor: "
55    << lastName << "", " << firstName << endl;
56
57 } // end destructor ~Employee

Output to show timing of destructors.
// Fig. 7.10: fig07_10.cpp
// Demonstrating composition--an object with member objects.
#include <iostream>

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

#include "employee1.h"  // Employee class definition

int main()
{
    Date birth( 7, 24, 1949 );
    Date hire( 3, 12, 1988 );
    Employee manager( "Bob", "Jones", birth, hire );

    cout << '
';
    manager.print();

    cout << "\nTest Date constructor with invalid values:\n";
    Date lastDayOff( 14, 35, 1994 );  // invalid month and day
    cout << endl;

    return 0;
}

// Create Date objects to pass to Employee constructor.
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Jones

Jones, Bob
Hired: 3/12/1988  Birth date: 7/24/1949

Test Date constructor with invalid values:
Month 14 invalid. Set to month 1.
Day 35 invalid. Set to day 1.
Date object constructor for date 1/1/1994

Date object destructor for date 1/1/1994
Employee object destructor: Jones, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949

Note two additional Date objects constructed; no output since default copy constructor used.

Destructor for date birth.
Destructor for Date object hire.
Destructor for Employee’s member object hireDate.
Destructor for Employee’s member object birthDate.
Destructor for Host object manager runs before destructors for member objects hireDate and birthDate.