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

Lecture 3: Classes

May 24, 2004
Classes
Structure Definitions

• Structures
  – Aggregate data types built using elements of other types

```c
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 ( -> ) for structure and class members via pointer to object
  - Print member *hour* of *timeObject*:

    ```
    cout << timeObject.hour;
    ``
    OR
    ```
    timePtr = &timeObject;
    cout << timePtr->hour;
    ```
  - *timePtr->hour* same as ( *timePtr ).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.
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
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:

```cpp
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
```

Class name becomes new type specifier.
Implementing a Time Abstract Data Type with a class

- **Member functions defined outside class**
  - Binary scope resolution operator (::)
    
    ```cpp
    ReturnType ClassName::MemberFunctionName()
    {
      ...
    }
    ```

- **Member functions defined inside class**
  - Do not need scope resolution operator, class name
  - Compiler attempts `inline`
    - Outside class, inline explicitly with keyword `inline`
```cpp
// 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

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

// 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" );
}

int main()
{
    Time t; // instantiate object t of class Time
70 // output Time object t's initial values
71 cout << "The initial universal time is ";
72 t.printUniversal(); // 00:00:00
73 cout << "\nThe initial standard time is ";
74 t.printStandard(); // 12:00:00 AM
75 t.setTime( 13, 27, 6 ); // change time
76 // output Time object t's new values
77 cout << "\nUniversal time after setTime is ";
78 t.printUniversal(); // 13:27:06
79 cout << "\nStandard time after setTime is ";
80 t.printStandard(); // 1:27:06 PM
81 t.setTime( 99, 99, 99 ); // attempt invalid settings
82 // output t's values after specifying invalid values
83 cout << "\n\nAfter attempting invalid settings:";
84 cout << "\nUniversal time: ";
85 t.printUniversal(); // 00:00:00
86 cout << "\nUniversal time: ";
87 t.printUniversal(); // 00:00:00
88 cout << "\nUniversal time: ";
89 t.printUniversal(); // 00:00:00
cout << "\nStandard time: ";
94 t.printStandard(); // 12:00:00 AM
95 cout << endl;
96
97 return 0;
98
99 } // 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 (~)
  - 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 structs
  - 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
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
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
Controlling Access to Members

- **Class member access**
  - Default *private*
  - Explicitly set to *private, public, protected*

- **struct member access**
  - Default *public*
  - Explicitly set to *private, public, protected*

- **Access to class’s *private* data**
  - Controlled with access functions (accessor methods)
    - Get function
      - Read *private* data
    - Set function
      - Modify *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

// Fig. 6.10: salesp.cpp
// Member functions for class SalesPerson.
#include <iostream>

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

#include <iomanip>

using std::setprecision;

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

// initialize elements of array sales to 0.0
SalesPerson::SalesPerson()
{
    for ( int i = 0; i < 12; i++ )
        sales[ i ] = 0.0;
}

// end SalesPerson constructor
// get 12 sales figures from the user at the keyboard
void SalesPerson::getSalesFromUser()
{
    double salesFigure;

    for ( int i = 1; i <= 12; i++ ) {
        cout << "Enter sales amount for month " << i << ": ";
        cin >> salesFigure;
        setSales( i, salesFigure );
    } // end for

} // end function getSalesFromUser

// set one of the 12 monthly sales figures; function subtracts
// one from month value for proper subscript
void SalesPerson::setSales( int month, double amount )
{
    // test for valid month and amount values
    if ( month >= 1 && month <= 12 && amount > 0 )
        sales[ month - 1 ] = amount; // adjust for subscripts 0-11

    else // invalid month or amount value
        cout << "Invalid month or sales figure" << endl;

    Set access function performs validity checks.
// print total annual sales (with help of utility function)
void SalesPerson::printAnnualSales()
{
    cout << setprecision( 2 ) << fixed
         << "\nThe total annual sales are: $" << setprecision( 2 ) << fixed
         << totalAnnualSales() << endl; // call utility function
}

// private utility function to total annual sales
double SalesPerson::totalAnnualSales()
{
    double total = 0.0; // initialize total
    for ( int i = 0; i < 12; i++ ) // summarize sales results
        total += sales[ i ];
    return total;
}

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
    
    ```
    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

Default constructor specifying all arguments.
// 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 "time2.h"

// include definition of class Time from 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
}

// Constructor calls \texttt{setTime} to validate passed (or default) values.
23 // set new Time value using universal time, perform validity
24 // checks on the data values and 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
// 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  ";
t2.printUniversal(); // 02:00:00
cout << "\n ";
t2.printStandard(); // 02:00 AM

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

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

cout << "\n\nall invalid values specified:\n  ";
t5.printUniversal(); // 00:00:00
cout << "\n ";
t5.printStandard(); // 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();             // destructor
private:
  int objectID;
  char *message;
}; // end class CreateAndDestroy
#endif
// 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;
}

Output message to demonstrate timing of constructor function calls.
23 // destructor
24 CreateAndDestroy::~CreateAndDestroy()
25 {
26     // the following line is for pedagogic purposes only
27     cout << ( objectID == 1 || objectID == 6 ? "\n" : """);
28     cout << "Object " << objectID << " destructor runs "
29     << message << endl;
30 }
31
32 } // end ~CreateAndDestroy destructor

Output message to demonstrate timing of destructor function calls.
// 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;

} // end main

// 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)" );

cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;

} // end function create

Create local automatic objects.

Create local automatic object.

Create local automatic object in function.

Create static local object in function.

Create local automatic object in function.

Create local automatic object in function.
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

};
#endif // TIME3_H
25 void printUniversal(); // output universal-time format
26 void printStandard(); // output standard-time format
27
28 private:
29   int hour;   // 0 – 23 (24-hour clock format)
30   int minute; // 0 – 59
31   int second; // 0 – 59
32
33 }; // end class Time
34
35 #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;
}

Call set functions to perform validity checking.

Set functions perform validity checks before modifying data.
// set second value
void Time::setSecond( int s )
{
    second = ( s >= 0 && s < 60 ) ? s : 0;
}

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

// return minute value
int Time::getMinute()
{
    return minute;
}
// 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 << "\nminute + 1: ";
        tt.printStandard();
    }
}

} // end function incrementMinutes

Using get functions to read data and set functions to modify data.
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
```cpp
// 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
```

Function to demonstrate effects of returning reference to `private` data member.
25 // return hour value
26 int Time::getHour()
27 {
28     return hour;
29
30 } // end function getHour
31
32 // POOR PROGRAMMING PRACTICE:
33 // Returning a reference to a private data member.
34 int &Time::badSetHour( int hh )
35 {
36     hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
37     return hour;  // DANGEROUS reference return
38 }
39 // 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();
}
26  // Dangerous: Function call that returns
27  // a reference can be used as an lvalue!
28  t.badSetHour( 12 ) = 74;
29
30  cout << "\n\n************************************************
          POOR PROGRAMMING PRACTICE!!!!!!!
          badSetHour as an lvalue, Hour: 74
          ************************************************\n"
31  << t.getHour() << endl;
32
33  return 0;
34
35 } // end main

Hour before modification: 20
Hour after modification: 30

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

Can use function call as lvalue to set invalid value.

Returning reference allowed invalid setting of private
data member hour.
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


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
    ```cpp
    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
**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;
    }
    void print() const;   // prints count and increment

private:
    int count;
    int increment;
};

void Increment::print() const
{
    cout << "Count = " << count << " Increment = " << increment << endl;
}
private:
int count;
const int increment; // const data member
}

// 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;
}

Member initializer list 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).
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

 ifndef
// 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.\n";
    }

    year = yr;             // should validate yr
    day = checkDay( dy );  // validate the day
1 // Fig. 7.8: employee1.h
2 // Employee class definition.
3 // Member functions defined in employee1.cpp.
4 #ifndef EMPLOYEE1_H
5 #define EMPLOYEE1_H
6
7 // include Date class definition from date1.h
8 #include "date1.h"
9
10 class Employee {
11
12 public:
13     Employee(
14         const char *, const char *, const Date &, const Date &);
15
16     void print() const;
17     ~Employee(); // provided to confirm destruction order
18
19 private:
20     char firstName[ 25 ];
21     char lastName[ 25 ];
22     const Date birthDate; // composition: member object
23     const Date hireDate;  // composition: member object
24
25 }; // 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
// constructor uses member initializer list to pass initializer
// values to constructors of member objects birthDate and
// hireDate [Note: This invokes the so-called "default copy
// constructor" which the C++ compiler provides implicitly.]
Employee::Employee( const char *first, const char *last,
               const Date &dateOfBirth, const Date &dateOfHire )
   : birthDate( dateOfBirth ),  // initialize birthDate
     hireDate( dateOfHire )     // initialize hireDate
{
   // copy first into firstName and be sure that it fits
   int length = strlen( first );
   length = ( length < 25 ? length : 24 );
   strncpy( firstName, first, length );
   firstName[ length ] = '\0';

   // copy last into lastName and be sure that it fits
   length = strlen( last );
   length = ( length < 25 ? length : 24 );
   strncpy( lastName, last, length );
   lastName[ length ] = '\0';

   // output Employee object to show when constructor is called
   cout << "Employee object constructor: "
        << firstName << ' ' << lastName << endl;
employee1.cpp

```cpp
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
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 << '\n';
    manager.print();

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

    return 0;
} // end main

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 \texttt{Date} objects constructed; no output since default copy constructor used.