Lecture 19:
Classes: A Deeper Look

Ioan Raicu (Guest Lecture by Martin Luessi)
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
April 28th, 2010

9.6 `Time` Class Case Study: Constructors with Default Arguments

- Like other functions, constructors can specify default arguments.
- The default arguments to the constructor ensure that, even if no values are provided in a constructor call, the constructor still initializes the data members to maintain the `Time` object in a consistent state.
- A constructor that defaults all its arguments is also a default constructor—i.e., a constructor that can be invoked with no arguments.
- There can be at most one default constructor per class.

9.6 Time Class Case Study: Constructors with Default Arguments

```
1  // Fig. 9.8: Time.h
2  // Time class containing a constructor with default arguments.
3  // Member functions defined in Time.cpp.
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME_H
7  #define TIME_H
8
9  // Time abstract data type definition
10 class Time
11 {
12 public:
13    Time( int = 0, int = 0, int = 0 ); // default constructor
14
15    // set functions
16    void setTime( int, int, int ); // set hour, minute, second
17    void setHour( int ); // set hour (after validation)
18    void setMinute( int ); // set minute (after validation)
19    void setSecond( int ); // set second (after validation)
20
```

Fig. 9.8 | `Time` class containing a constructor with default arguments. (Part 1 of 2.)

9.6 Time Class Case Study: Constructors with Default Arguments

```
21      // get functions
22      int getHour(); // return hour
23      int getMinute(); // return minute
24      int getSecond(); // return second
25
26      void printUniversal(); // output time in universal-time format
27      void printStandard(); // output time in standard-time format
28   private:
29      int hour; // 0 - 23 (24-hour clock format)
30      int minute; // 0 - 59
31      int second; // 0 - 59
32   }; // end class Time
33
34   #endif
```

Fig. 9.8 | `Time` class containing a constructor with default arguments. (Part 2 of 2.)

9.6 Time Class Case Study: Constructors with Default Arguments

```
1  // Fig. 9.9: Time.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4  #include <iomanip>
5  #include "Time.h" // include definition of class Time from Time.h
6  using namespace std;
7
8  // Time constructor initializes each data member to zero;
9  // ensures that Time objects start in a consistent state
10 Time::Time( int hr, int min, int sec )
11 {
12    setTime( hr, min, sec ); // validate and set time
13 } // end Time constructor
14
15 // set new Time value using universal time; ensure that
16 // the data remains consistent by setting invalid values to zero
17 void Time::setTime( int h, int m, int s )
18 {
19    setHour( h ); // set private field hour
20    setMinute( m ); // set private field minute
21    setSecond( s ); // set private field second
22 } // end function setTime
```

Fig. 9.9 | `Time` class member-function definitions including a constructor that takes arguments. (Part 1 of 4.)

9.6 Time Class Case Study: Constructors with Default Arguments

```
23
24  // set hour value
25  void Time::setHour( int h )
26  {
27     hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
28  } // end function setHour
29
30  // set minute value
31  void Time::setMinute( int m )
32  {
33     minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
34  } // end function setMinute
35
36  // set second value
37  void Time::setSecond( int s )
38  {
39     second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
40  } // end function setSecond
41
```

Fig. 9.9 | `Time` class member-function definitions including a constructor that takes arguments. (Part 2 of 4.)

9.6 Time Class Case Study: Constructors with Default Arguments

```
42  // return hour value
43  int Time::getHour()
44  {
45     return hour;
46  } // end function getHour
47
48  // return minute value
49  int Time::getMinute()
50  {
51     return minute;
52  } // end function getMinute
53
54  // return second value
55  int Time::getSecond()
56  {
57     return second;
58  } // end function getSecond
59
```

Fig. 9.9 | `Time` class member-function definitions including a constructor that takes arguments. (Part 3 of 4.)

9.6 Time Class Case Study: Constructors with Default Arguments

```
60  // print Time in universal-time format (HH:MM:SS)
61  void Time::printUniversal()
62  {
63     cout << setfill( '0' ) << setw( 2 ) << getHour() << ":"
64        << setw( 2 ) << getMinute() << ":" << setw( 2 ) << getSecond();
65  } // end function printUniversal
66
67  // print Time in standard-time format (HH:MM:SS AM or PM)
68  void Time::printStandard()
69  {
70     cout << ( ( getHour() == 0 || getHour() == 12 ) ? 12 : getHour() % 12 )
71        << ":" << setfill( '0' ) << setw( 2 ) << getMinute()
72        << ":" << setw( 2 ) << getSecond() << ( hour < 12 ? " AM" : " PM" );
73  } // end function printStandard
```

Fig. 9.9 | `Time` class member-function definitions including a constructor that takes arguments. (Part 4 of 4.)

9.6 `Time` Class Case Study: Constructors with Default Arguments (cont.)

- Calling `setHour`, `setMinute` and `setSecond` from the constructor may be slightly more efficient because the extra call to `setTime` would be eliminated.
- Similarly, copying the code from lines 27, 33 and 39 into constructor would eliminate the overhead of calling `setTime`, `setHour`, `setMinute` and `setSecond`.
- This would make maintenance of this class more difficult.
  - If the implementations of `setHour`, `setMinute` and `setSecond` were to change, the implementation of any member function that duplicates lines 27, 33 and 39 would have to change accordingly.
- Calling `setTime` and having `setTime` call `setHour`, `setMinute` and `setSecond` enables us to limit the changes to the corresponding *set function*.
  - Reduces the likelihood of errors when altering the implementation.

9.6 Time Class Case Study: Constructors with Default Arguments (cont.)

Software Engineering Observation 9.9

If a member function of a class already provides all or part of the functionality required by a constructor (or other member function) of the class, call that member function from the constructor (or other member function). This simplifies the maintenance of the code and reduces the likelihood of an error if the implementation of the code is modified. As a general rule: Avoid repeating code.

9.6 Time Class Case Study: Constructors with Default Arguments (cont.)

```
1  // Fig. 9.10: fig09_10.cpp
2  // Demonstrating a default constructor for class Time.
3  #include <iostream>
4  #include "Time.h" // include definition of class Time from Time.h
5  using namespace std;
6
7  int main()
8  {
9     Time t1; // all arguments defaulted
10    Time t2( 2 ); // hour specified; minute and second defaulted
11    Time t3( 21, 34 ); // hour and minute specified; second defaulted
12    Time t4( 12, 25, 42 ); // hour, minute and second specified
13    Time t5( 27, 74, 99 ); // all bad values specified
14
15    cout << "Constructed with:\n\nt1: all arguments defaulted\n  ";
16    t1.printUniversal(); // 00:00:00
17    cout << "\n  ";
18    t1.printStandard(); // 12:00:00 AM
19
20    cout << "\n\nt2: hour specified; minute and second defaulted\n  ";
21    t2.printUniversal(); // 02:00:00
22    cout << "\n  ";
23    t2.printStandard(); // 2:00:00 AM
```

Fig. 9.10 | Constructor with default arguments. (Part 1 of 3.)

9.6 Time Class Case Study: Constructors with Default Arguments (cont.)

```
24
25      cout << "\n\nt3: hour and minute specified; second defaulted\n  ";
26      t3.printUniversal(); // 21:34:00
27      cout << "\n  ";
28      t3.printStandard(); // 9:34:00 PM
29
30      cout << "\n\nt4: hour, minute and second specified\n  ";
31      t4.printUniversal(); // 12:25:42
32      cout << "\n  ";
33      t4.printStandard(); // 12:25:42 PM
34
35      cout << "\n\nt5: all invalid values specified\n  ";
36      t5.printUniversal(); // 00:00:00
37      cout << "\n  ";
38      t5.printStandard(); // 12:00:00 AM
39      cout << endl;
40   } // end main
```

Fig. 9.10 | Constructor with default arguments. (Part 2 of 3.)

9.6 Time Class Case Study: Constructors with Default Arguments (cont.)

```
Constructed with:

t1: all arguments defaulted
  00:00:00
  12:00:00 AM

t2: hour specified; minute and second defaulted
  02:00:00
  2:00:00 AM

t3: hour and minute specified; second defaulted
  21:34:00
  9:34:00 PM

t4: hour, minute and second specified
  12:25:42
  12:25:42 PM

t5: all invalid values specified
  00:00:00
  12:00:00 AM
```

Fig. 9.10 | Constructor with default arguments. (Part 3 of 3.)

9.7 Destructors

- The name of the destructor for a class is the tilde character (~) followed by the class name.
- Often referred to with the abbreviation "dtor" in the literature.
- Called implicitly when an object is destroyed.
- *The destructor itself does not actually release the object's memory—it performs termination housekeeping before the object's memory is reclaimed, so the memory may be reused to hold new objects.*
- Receives no parameters and returns no value.
- May not specify a return type—not even `void`.
- A class may have only one destructor.
- A destructor must be `public`.
- If you do not explicitly provide a destructor, the compiler creates an "empty" destructor.

9.8 When Constructors and Destructors Are Called

- Constructors and destructors are called implicitly.
- The order in which these function calls occur depends on the order in which execution enters and leaves the scopes where the objects are instantiated.
- Generally, destructor calls are made in the reverse order of the corresponding constructor calls
  - The storage classes of objects can alter the order in which destructors are called.
- Constructors are called for objects defined in global scope before any other function (including `main`) in that file begins exe-cution (although the order of execution of global object constructors between files is not guaranteed).
  - The corresponding destructors are called when `main` terminates.

9.8 When Constructors and Destructors Are Called (cont.)

- Function `exit` forces a program to terminate immediately and does not execute the destructors of automatic objects.
- Function `abort` performs similarly to function `exit` but forces the program to terminate immediately, without allowing the destructors of any objects to be called.

9.8 When Constructors and Destructors Are Called (cont.)

- Constructors and destructors for automatic objects are called each time execution enters and leaves the scope of the object.
- Destructors are not called for automatic objects if the program terminates with a call to function `exit` or function `abort`.
- The constructor for a `static` local object is called only once, when execution first reaches the point where the object is defined—the corresponding destructor is called when `main` terminates or the program calls function `exit`.
- Global and `static` objects are destroyed in the reverse order of their creation.
- Destructors are not called for `static` objects if the program terminates with a call to function `abort`.

9.8 When Constructors and Destructors Are Called (cont.)

```
1  // Fig. 9.11: CreateAndDestroy.h
2  // CreateAndDestroy class definition.
3  // Member functions defined in CreateAndDestroy.cpp.
4  #include <string>
5  using namespace std;
6
7  #ifndef CREATE_H
8  #define CREATE_H
9
10 class CreateAndDestroy
11 {
12 public:
13    CreateAndDestroy( int, string ); // constructor
14    ~CreateAndDestroy(); // destructor
15 private:
16    int objectID; // ID number for object
17    string message; // message describing object
18 }; // end class CreateAndDestroy
19
20 #endif
```

Fig. 9.11 | `CreateAndDestroy` class definition.

9.8 When Constructors and Destructors Are Called (cont.)

```
1  // Fig. 9.12: CreateAndDestroy.cpp
2  // CreateAndDestroy class member-function definitions.
3  #include <iostream>
4  #include "CreateAndDestroy.h"// include CreateAndDestroy class definition
5  using namespace std;
6
7  // constructor
8  CreateAndDestroy::CreateAndDestroy( int ID, string messageString )
9  {
10    objectID = ID; // set object's ID number
11    message = messageString; // set object's descriptive message
12
13    cout << "Object " << objectID << "   constructor runs    "
14       << message << endl;
15 } // end CreateAndDestroy constructor
16
17 // destructor
18 CreateAndDestroy::~CreateAndDestroy()
19 {
20    // output newline for certain objects; helps readability
21    cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );
22
23    cout << "Object " << objectID << "   destructor runs    "
24       << message << endl;
25 } // end ~CreateAndDestroy destructor
```

Fig. 9.12 | CreateAndDestroy class member-function definitions.

Time Class Case Study: A Subtle Trap—Returning a Reference to a private Data Member

- A reference to an object is an alias for the name of the object and, hence, may be used on the left side of an assignment statement.
- In this context, the reference makes a perfectly acceptable *lvalue* that can receive a value.
- Unfortunately a `public` member function of a class can return a reference to a `private` data member of that class.
- Such a reference return actually makes a call to that member function an alias for the `private` data member!
  - The function call can be used in any way that the `private` data member can be used, including as an *lvalue* in an assignment statement
  - The same problem would occur if a pointer to the `private` data were to be returned by the function.
- If a function returns a `const` reference, that reference cannot be used as a modifiable *lvalue*.

Time Class Case Study: A Subtle Trap—Returning a Reference to a private Data Member

```
1  // Fig. 9.14: Time.h
2  // Time class declaration.
3  // Member functions defined in Time.cpp
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME_H
7  #define TIME_H
8
9  class Time
10 {
11 public:
12    Time( int = 0, int = 0, int = 0 );
13    void setTime( int, int, int );
14    int getHour();
15    int &badSetHour( int ); // DANGEROUS reference return
16 private:
17    int hour;
18    int minute;
19    int second;
20 }; // end class Time
21
22 #endif
```

Fig. 9.14 | `Time` class declaration.

Time Class Case Study: A Subtle Trap—Returning a Reference to a private Data Member

```
1  // Fig. 9.15: Time.cpp
2  // Time class member-function definitions.
3  #include "Time.h" // include definition of class Time
4
5  // constructor function to initialize private data; calls member function
6  // setTime to set variables; default values are 0 (see class definition)
7  Time::Time( int hr, int min, int sec )
8  {
9     setTime( hr, min, sec );
10 } // end Time constructor
11
12 // set values of hour, minute and second
13 void Time::setTime( int h, int m, int s )
14 {
15    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
16    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
17    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
18 } // end function setTime
19
20 // return hour value
21 int Time::getHour()
22 {
23    return hour;
24 } // end function getHour
```

Fig. 9.15 | Time class member-function definitions. (Part 1 of 2.)

Time Class Case Study: A Subtle Trap—Returning a Reference to a private Data Member

```
25
26 // POOR PRACTICE: Returning a reference to a private data member.
27 int &Time::badSetHour( int hh )
28 {
29    hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
30    return hour; // DANGEROUS reference return
31 } // end function badSetHour
```

Fig. 9.15 | `Time` class member-function definitions. (Part 2 of 2.)

Time Class Case Study: A Subtle Trap—Returning a Reference to a private Data Member

```
1  // Fig. 9.16: fig09_16.cpp
2  // Demonstrating a public member function that
3  // returns a reference to a private data member.
4  #include <iostream>
5  #include "Time.h" // include definition of class Time
6  using namespace std;
7
8  int main()
9  {
10    Time t; // create Time object
11
12    // initialize hourRef with the reference returned by badSetHour
13    int &hourRef = t.badSetHour( 20 ); // 20 is a valid hour
14
15    cout << "Valid hour before modification: " << hourRef;
16    hourRef = 30; // use hourRef to set invalid value in Time object t
17    cout << "\nInvalid hour after modification: " << t.getHour();
18
19    // Dangerous: Function call that returns
20    // a reference can be used as an lvalue!
21    t.badSetHour( 12 ) = 74; // assign another invalid value to hour
22
```

Fig. 9.16 | Returning a reference to a `private` data member. (Part 1 of 2.)

Time Class Case Study: A Subtle Trap—Returning a Reference to a private Data Member

```
23      cout << "\n\n************************************************\n"
24         << "POOR PROGRAMMING PRACTICE!!!!!!!!\n"
25         << "t.badSetHour( 12 ) as an lvalue, invalid hour: "
26         << t.getHour()
27         << "\n************************************************" << endl;
28   } // end main
```

```
Valid hour before modification: 20
Invalid hour after modification: 30

************************************************
POOR PROGRAMMING PRACTICE!!!!!!!!
t.badSetHour( 12 ) as an lvalue, invalid hour: 74
************************************************
```

Fig. 9.16 | Returning a reference to a `private` data member. (Part 2 of 2.)

Time Class Case Study: A Subtle Trap—Returning a Reference to a private Data Member

Error-Prevention Tip 9.4

Returning a reference or a pointer to a `private` *data member breaks the encapsulation of the class and makes the client code dependent on the representation of the class's data; this is a dangerous practice that should be avoided.*

9.9 Default Memberwise Assignment

- The assignment operator (=) can be used to assign an object to another object of the same type.
- By default, such assignment is performed by memberwise assignment
  - Each data member of the object on the right of the assignment operator is assigned individually to the same data member in the object on the left of the assignment operator.
- [*Caution:* Memberwise assignment can cause serious problems when used with a class whose data members contain pointers to dynamically allocated memory; we discuss these problems in Chapter 11 and show how to deal with them.]

9.9 Default Memberwise Assignment

```
1  // Fig. 9.17: Date.h
2  // Date class declaration.  Member functions are defined in Date.cpp.
3
4  // prevent multiple inclusions of header file
5  #ifndef DATE_H
6  #define DATE_H
7
8  // class Date definition
9  class Date
10 {
11 public:
12    Date( int = 1, int = 1, int = 2000 ); // default constructor
13    void print();
14 private:
15    int month;
16    int day;
17    int year;
18 }; // end class Date
19
20 #endif
```

Fig. 9.17 | `Date` class declaration.

9.9 Default Memberwise Assignment

```
1  // Fig. 9.18: Date.cpp
2  // Date class member-function definitions.
3  #include <iostream>
4  #include "Date.h" // include definition of class Date from Date.h
5  using namespace std;
6
7  // Date constructor (should do range checking)
8  Date::Date( int m, int d, int y )
9  {
10    month = m;
11    day = d;
12    year = y;
13 } // end constructor Date
14
15 // print Date in the format mm/dd/yyyy
16 void Date::print()
17 {
18    cout << month << '/' << day << '/' << year;
19 } // end function print
```

Fig. 9.18 | `Date` class member-function definitions.

9.9 Default Memberwise Assignment

```
1  // Fig. 9.19: fig09_19.cpp
2  // Demonstrating that class objects can be assigned
3  // to each other using default memberwise assignment.
4  #include <iostream>
5  #include "Date.h" // include definition of class Date from Date.h
6  using namespace std;
7
8  int main()
9  {
10    Date date1( 7, 4, 2004 );
11    Date date2; // date2 defaults to 1/1/2000
12
13    cout << "date1 = ";
14    date1.print();
15    cout << "\ndate2 = ";
16    date2.print();
17
18    date2 = date1; // default memberwise assignment
19
20    cout << "\n\nAfter default memberwise assignment, date2 = ";
21    date2.print();
22    cout << endl;
23 } // end main
```

Fig. 9.19 | Default memberwise assignment. (Part 1 of 2.)

9.9 Default Memberwise Assignment

```
date1 = 7/4/2004
date2 = 1/1/2000

After default memberwise assignment, date2 = 7/4/2004
```

Fig. 9.19 | Default memberwise assignment. (Part 2 of 2.)

9.9 Default Memberwise Assignment (cont.)

- Objects may be passed as function arguments and may be returned from functions.
- Such passing and returning is performed using pass-by-value by default—a copy of the object is passed or returned.
  - C++ creates a new object and uses a copy constructor to copy the original object's values into the new object.
- For each class, the compiler provides a default copy constructor that copies each member of the original object into the corresponding member of the new object.
  - Copy constructors can cause serious problems when used with a class whose data members contain pointers to dynamically allocated memory.
- Chapter 11 discusses customized copy constructors.

Questions