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
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

```cpp
// Fig. 9.8: Time.h
// Time class containing a constructor with default arguments.
// Member functions defined in Time.cpp.

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

// Time abstract data type definition
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 (after validation)
    void setMinute( int ); // set minute (after validation)
    void setSecond( int ); // set second (after validation)

Fig. 9.8 | Time class containing a constructor with default arguments. (Part I of 2.)
```
9.6 Time Class Case Study: Constructors with Default Arguments

```c
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

```cpp
// Fig. 9.9: Time.cpp
// Member-function definitions for class Time.
#include <iostream>
#include <iomanip>
#include "Time.h" // include definition of class Time from Time.h
using namespace std;

// Time constructor initializes each data member to zero;
// ensures that 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; ensure that
// the data remains consistent by setting invalid values to zero
void Time::setTime( int h, int m, int s )
{
    setHour( h ); // set private field hour
    setMinute( m ); // set private field minute
    setSecond( s ); // set private field second
} // 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

Fig. 9.9 | Time class member-function definitions including a constructor that takes arguments. (Part 2 of 4.)
// return hour value
int Time::getHour()
{
    return hour;
} // end function getHour

// return minute value
int Time::getMinute()
{
    return minute;
} // end function getMinute

// return second value
int Time::getSecond()
{
    return second;
} // end function getSecond

**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

```cpp
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

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.)
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.
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.
```cpp
// Fig. 9.10: fig09_10.cpp
// Demonstrating a default constructor for class Time.
#include <iostream>
#include "Time.h"  // include definition of class Time from Time.h
using namespace std;

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

    cout << "Constructed with:\n\nt1: all arguments defaulted\n  ";
t1.printUniversal();  // 00:00:00
    cout << "\n  ";
t1.printStandard();  // 12:00:00 AM

    cout << "\n\nt2: hour specified; minute and second defaulted\n  ";
t2.printUniversal();  // 02:00:00
    cout << "\n  ";
t2.printStandard();  // 2:00:00 AM
}
```

**Fig. 9.10**  |  Constructor with default arguments. (Part I of 3.)
9.6 Time Class Case Study: Constructors with Default Arguments (cont.)

```cpp
24 cout << "\n\nt3: hour and minute specified; second defaulted\n  ";
25 t3.printUniversal(); // 21:34:00
26 cout << "\n  ";
27 t3.printStandard(); // 9:34:00 PM
28 
29 cout << "\n\nt4: hour, minute and second specified\n  ";
30 t4.printUniversal(); // 12:25:42
31 cout << "\n  ";
32 t4.printStandard(); // 12:25:42 PM
33 
34 cout << "\n\nt5: all invalid values specified\n  ";
35 t5.printUniversal(); // 00:00:00
36 cout << "\n  ";
37 t5.printStandard(); // 12:00:00 AM
38 cout << "\n  ";
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.)
The name of the destructor for a class is the \texttt{tilde} character (\texttt{~}) followed by the class name.

Often referred to with the abbreviation “\texttt{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 \texttt{void}.

A class may have only one destructor.

A destructor must be \texttt{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 execution (although the order of execution of global object constructors between files is not guaranteed).
  – The corresponding destructors are called when `main` terminates.
• 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.
• 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`.


```cpp
// Fig. 9.11: CreateAndDestroy.h
// CreateAndDestroy class definition.
// Member functions defined in CreateAndDestroy.cpp.
#include <string>
using namespace std;

#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy
{
public:
    CreateAndDestroy( int, string ); // constructor
    ~CreateAndDestroy(); // destructor

private:
    int objectId; // ID number for object
    string message; // message describing object
}; // end class CreateAndDestroy

#endif
```

**Fig. 9.11**  |  CreateAndDestroy class definition.
9.8 When Constructors and Destructors Are Called (cont.)

// Fig. 9.12: CreateAndDestroy.cpp
// CreateAndDestroy class member-function definitions.
#include <iostream>
#include "CreateAndDestroy.h" // include CreateAndDestroy class definition
using namespace std;

// constructor
CreateAndDestroy::CreateAndDestroy( int ID, string messageString )
{
    objectID = ID; // set object's ID number
    message = messageString; // set object's descriptive message

cout << "Object " << objectID << " constructor runs "
    << message << endl;
} // end CreateAndDestroy constructor

// destructor
CreateAndDestroy::~CreateAndDestroy()
{
    // output newline for certain objects; helps readability
    cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );

cout << "Object " << objectID << " destructor runs "
    << message << endl;
} // end ~CreateAndDestroy destructor
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 \textit{lvalue} that can receive a value. Unfortunately a \texttt{public} member function of a class can return a reference to a \texttt{private} data member of that class. Such a reference return actually makes a call to that member function an alias for the \texttt{private} data member!

- The function call can be used in any way that the \texttt{private} data member can be used, including as an \textit{lvalue} in an assignment statement
- The same problem would occur if a pointer to the \texttt{private} data were to be returned by the function.

If a function returns a \texttt{const} reference, that reference cannot be used as a modifiable \textit{lvalue}. 

©1992-2010 by Pearson Education, Inc. All Rights Reserved.
// Fig. 9.14: Time.h
// Time class declaration.
// Member functions defined in Time.cpp

// prevent multiple inclusions of header file
#ifndef TIME_H
#define TIME_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. 9.14  |  Time class declaration.
Time Class Case Study: A Subtle Trap—
Returning a Reference to a private Data Member

```cpp
// Fig. 9.15: Time.cpp
// Time class member-function definitions.
#include "Time.h" // include definition of class Time

// 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; // validate hour
    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
} // end function setTime

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

©1992-2010 by Pearson Education, Inc. All Rights Reserved.
// POOR 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. 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

```cpp
// Fig. 9.16: fig09_16.cpp
// Demonstrating a public member function that returns a reference to a private data member.
#include <iostream>
#include "Time.h" // include definition of class Time
using namespace std;

int main()
{
    Time t; // create Time object

    int &hourRef = t.badSetHour(20); // 20 is a valid hour

    cout << "Valid hour before modification: " << hourRef;
    hourRef = 30; // use hourRef to set invalid value in Time object t
    cout << "\nInvalid hour after modification: " << t.getHour();

    // Dangerous: Function call that returns a reference can be used as an lvalue!
    t.badSetHour(12) = 74; // assign another invalid value to hour

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

©1992-2010 by Pearson Education, Inc. All Rights Reserved.
cout << "\n\n******************************************************************************\n" << "POOR PROGRAMMING PRACTICE!!!!!!!\n" << "t.badSetHour( 12 ) as an lvalue, invalid hour: " << t.getHour() << "\n***********************************************************************" << endl;

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.)
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.
• 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.]
// Fig. 9.17: Date.h
// Date class declaration. Member functions are defined in Date.cpp.

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

// class Date definition
class Date
{
public:
    Date( int = 1, int = 1, int = 2000 ); // default constructor
    void print();
private:
    int month;
    int day;
    int year;
}; // end class Date

#endif
// Fig. 9.18: Date.cpp
// Date class member-function definitions.
#include <iostream>
#include "Date.h" // include definition of class Date from Date.h
using namespace std;

// Date constructor (should do range checking)
Date::Date( int m, int d, int y )
{
    month = m;
    day = d;
    year = y;
} // end constructor Date

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

Fig. 9.18 | Date class member-function definitions.
// Fig. 9.19: fig09_19.cpp
// Demonstrating that class objects can be assigned
// to each other using default memberwise assignment.
#include <iostream>
#include "Date.h" // include definition of class Date from Date.h
using namespace std;

int main()
{
    Date date1( 7, 4, 2004 );
    Date date2; // date2 defaults to 1/1/2000

cout << "date1 = ";
date1.print();
cout << "\ndate2 = ";
date2.print();

date2 = date1; // default memberwise assignment

cout << "\n\nAfter default memberwise assignment, date2 = ";
date2.print();
cout << endl;
} // end main

Fig. 9.19  |  Default memberwise assignment. (Part 1 of 2.)
<table>
<thead>
<tr>
<th>date1</th>
<th>7/4/2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>date2</td>
<td>1/1/2000</td>
</tr>
</tbody>
</table>

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

**Fig. 9.19** Default memberwise assignment. (Part 2 of 2.)
• 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.