

Computer Programming

Chapter 10: Classes A Deeper Look

Hung-Yu Wei

Department of Electrical Engineering

National Taiwan University

Chapter 10 Overview

- Constant
 - Constant object
 - Constant data member
- Object as member of a class
- Friend
- this
- Static class member

const

- const objects and const member functions
 - prevent modifications of objects and enforce the principle of least privilege.
- Syntax – declare a constant object
 - **const** **ClassName** **Object Name;**
 - Example
 - `const Triangle myFirstTriangle;`
- Syntax – define a constant member function
 - **ReturnType** **FunctionName** **(Input)** **const**
 - Example
 - `int getHour() const`
- Syntax error!
 - When a constant object calls a non-constant member function

const member function

- Cannot modify data member
- Cannot access a non-const object
- Cannot access a non-const member function
- **const** keyword in both
 - Function prototype
 - Function implementation

```
1 // Fig. 10.1: Time.h
2 // Time class definition with const member functions.
3 // Member functions defined in Time.cpp.
4 #ifndef TIME_H
5 #define TIME_H
6
7 class Time
8 {
9 public:
10    Time( int = 0, int = 0, int = 0 ); // default constructor
11
12    // set functions
13    void setTime( int, int, int ); // set time
14    void setHour( int ); // set hour
15    void setMinute( int ); // set minute
16    void setSecond( int ); // set second
17
18    // get functions (normally declared const)
19    int getHour() const; // return hour
20    int getMinute() const; // return minute
21    int getSecond() const; // return second
22
```

Fig. 10.1 | Time class definition with const member functions. (Part I of 2.)

```
23 // print functions (normally declared const)
24 void printUniversal() const; // print universal time
25 void printStandard(); // print standard time (should be const)
26 private:
27     int hour; // 0 - 23 (24-hour clock format)
28     int minute; // 0 - 59
29     int second; // 0 - 59
30 }; // end class Time
31
32 #endif
```

Fig. 10.1 | Time class definition with const member functions. (Part 2 of 2.)

```
1 // Fig. 10.2: Time.cpp
2 // Time class member-function definitions.
3 #include <iostream>
4 #include <iomanip>
5 #include "Time.h" // include definition of class Time
6 using namespace std;
7
8 // constructor function to initialize private data;
9 // calls member function setTime to set variables;
10 // default values are 0 (see class definition)
11 Time::Time( int hour, int minute, int second )
12 {
13     setTime( hour, minute, second );
14 } // end Time constructor
15
16 // set hour, minute and second values
17 void Time::setTime( int hour, int minute, int second )
18 {
19     setHour( hour );
20     setMinute( minute );
21     setSecond( second );
22 } // end function setTime
```

Fig. 10.2 | Time class member-function definitions. (Part I of 4.)

```
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
42 // return hour value
43 int Time::getHour() const // get functions should be const
44 {
45     return hour;
46 } // end function getHour
```

Fig. 10.2 | Time class member-function definitions. (Part 2 of 4.)

```
47
48 // return minute value
49 int Time::getMinute() const
50 {
51     return minute;
52 } // end function getMinute
53
54 // return second value
55 int Time::getSecond() const
56 {
57     return second;
58 } // end function getSecond
59
60 // print Time in universal-time format (HH:MM:SS)
61 void Time::printUniversal() const
62 {
63     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
64         << setw( 2 ) << minute << ":" << setw( 2 ) << second;
65 } // end function printUniversal
66
```

Fig. 10.2 | Time class member-function definitions. (Part 3 of 4.)

```
67 // print Time in standard-time format (HH:MM:SS AM or PM)
68 void Time::printStandard() // note lack of const declaration
69 {
70     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
71         << ":" << setfill( '0' ) << setw( 2 ) << minute
72         << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
73 } // end function printStandard
```

Fig. 10.2 | Time class member-function definitions. (Part 4 of 4.)

```
1 // Fig. 10.3: fig10_03.cpp
2 // Attempting to access a const object with non-const member functions.
3 #include "Time.h" // include Time class definition
4
5 int main()
6 {
7     Time wakeUp( 6, 45, 0 ); // non-constant object
8     const Time noon( 12, 0, 0 ); // constant object
9
10            // OBJECT      MEMBER FUNCTION
11     wakeUp.setHour( 18 ); // non-const    non-const
12
13     noon.setHour( 12 ); // const        non-const
14
15     wakeUp.getHour();   // non-const    const
16
17     noon.getMinute();  // const        const
18     noon.printUniversal(); // const       const
19
20     noon.printStandard(); // const       non-const
21 } // end main
```

Fig. 10.3 | const objects and const member functions. (Part I of 2.)

Microsoft Visual C++ compiler error messages:

```
C:\cpphttp7_examples\ch10\Fig10_01_03\fig10_03.cpp(13) : error C2662:  
    'Time::setHour' : cannot convert 'this' pointer from 'const Time' to  
    'Time &'  
        Conversion loses qualifiers  
C:\cpphttp7_examples\ch10\Fig10_01_03\fig10_03.cpp(20) : error C2662:  
    'Time::printStandard' : cannot convert 'this' pointer from 'const Time' to  
    'Time &'  
        Conversion loses qualifiers
```

GNU C++ compiler error messages:

```
fig10_03.cpp:13: error: passing 'const Time' as 'this' argument of  
    'void Time::setHour(int)' discards qualifiers  
fig10_03.cpp:20: error: passing 'const Time' as 'this' argument of  
    'void Time::printStandard()' discards qualifiers
```

Fig. 10.3 | const objects and const member functions. (Part 2 of 2.)

[concept]member initializer

- Initialize data member
 - Constant data member
 - Non-constant data member
- Within constructor
- Syntax
 - : dataMember1 (initialValue1),
dataMember2 (initialValue2)

```
1 // Fig. 10.4: Increment.h
2 // Definition of class Increment.
3 #ifndef INCREMENT_H
4 #define INCREMENT_H
5
6 class Increment
7 {
8 public:
9     Increment( int c = 0, int i = 1 ); // default constructor
10
11    // function addIncrement definition
12    void addIncrement()
13    {
14        count += increment;
15    } // end function addIncrement
16
17    void print() const; // prints count and increment
18 private:
19    int count;
20    const int increment; // const data member
21 }; // end class Increment
22
23 #endif
```

Fig. 10.4 | Increment class definition containing non-const data member count and const data member increment.

```
1 // Fig. 10.5: Increment.cpp
2 // Member-function definitions for class Increment demonstrate using a
3 // member initializer to initialize a constant of a built-in data type.
4 #include <iostream>
5 #include "Increment.h" // include definition of class Increment
6 using namespace std;
7
8 // constructor
9 Increment::Increment( int c, int i )
10    : count( c ), // initializer for non-const member
11      increment( i ) // required initializer for const member
12 {
13     // empty body
14 } // end constructor Increment
15
16 // print count and increment values
17 void Increment::print() const
18 {
19     cout << "count = " << count << ", increment = " << increment << endl;
20 } // end function print
```

Fig. 10.5 | Member initializer used to initialize a constant of a built-in data type.

```
1 // Fig. 10.6: fig10_06.cpp
2 // Program to test class Increment.
3 #include <iostream>
4 #include "Increment.h" // include definition of class Increment
5 using namespace std;
6
7 int main()
8 {
9     Increment value( 10, 5 );
10
11    cout << "Before incrementing: ";
12    value.print();
13
14    for ( int j = 1; j <= 3; j++ )
15    {
16        value.addIncrement();
17        cout << "After increment " << j << ": ";
18        value.print();
19    } // end for
20 } // end main
```

Fig. 10.6 | Invoking an `Increment` object's `print` and `addIncrement` member functions.

const data member

- Constant data member must be initialized using “member initializer”

```
18 private:  
19     int count;  
20     const int increment; // const data member
```

```
8 // constructor; constant member 'increment' is not initialized  
9 Increment::Increment( int c, int i )  
10 {  
11     count = c; // allowed because count is not constant  
12     increment = i; // ERROR: Cannot modify a const object  
13 } // end constructor Increment
```

OK

```
9 Increment::Increment( int c, int i )
10      : count( c ), // initializer for non-const member
11          increment( i ) // required initializer for const member
12 {
13     // empty body
14 } // end constructor Increment
```

Error

```
8 // constructor; constant member 'increment' is not initialized
9 Increment::Increment( int c, int i )
10 {
11     count = c; // allowed because count is not constant
12     increment = i; // ERROR: Cannot modify a const object
13 } // end constructor Increment
```

10.3 Objects as member of a class

- A class can have objects of other classes as members
 - Known as “composition”
- Class is useful !

```
1 // Fig. 10.10: Date.h
2 // Date class definition; Member functions defined in Date.cpp
3 #ifndef DATE_H
4 #define DATE_H
5
6 class Date
7 {
8 public:
9     static const int monthsPerYear = 12; // number of months in a year
10    Date( int = 1, int = 1, int = 1900 ); // default constructor
11    void print() const; // print date in month/day/year format
12    ~Date(); // provided to confirm destruction order
13 private:
14     int month; // 1-12 (January-December)
15     int day; // 1-31 based on month
16     int year; // any year
17
18     // utility function to check if day is proper for month and year
19     int checkDay( int ) const;
20 }; // end class Date
21
22 #endif
```

Fig. 10.10 | Date class definition.

```
1 // Fig. 10.11: Date.cpp
2 // Date class member-function definitions.
3 #include <iostream>
4 #include "Date.h" // include Date class definition
5 using namespace std;
6
7 // constructor confirms proper value for month; calls
8 // utility function checkDay to confirm proper value for day
9 Date::Date( int mn, int dy, int yr )
10 {
11     if ( mn > 0 && mn <= monthsPerYear ) // validate the month
12         month = mn;
13     else
14     {
15         month = 1; // invalid month set to 1
16         cout << "Invalid month (" << mn << ") set to 1.\n";
17     } // end else
18
19     year = yr; // could validate yr
20     day = checkDay( dy ); // validate the day
21
22     // output Date object to show when its constructor is called
23     cout << "Date object constructor for date ";
```

Fig. 10.11 | Date class member-function definitions. (Part I of 3.)

```
24     print();
25     cout << endl;
26 } // end Date constructor
27
28 // print Date object in form month/day/year
29 void Date::print() const
30 {
31     cout << month << '/' << day << '/' << year;
32 } // end function print
33
34 // output Date object to show when its destructor is called
35 Date::~Date()
36 {
37     cout << "Date object destructor for date ";
38     print();
39     cout << endl;
40 } // end ~Date destructor
41
42 // utility function to confirm proper day value based on
43 // month and year; handles leap years, too
44 int Date::checkDay( int testDay ) const
45 {
46     static const int daysPerMonth[ monthsPerYear + 1 ] =
47         { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Fig. 10.11 | Date class member-function definitions. (Part 2 of 3.)

```
48
49     // determine whether testDay is valid for specified month
50     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
51         return testDay;
52
53     // February 29 check for leap year
54     if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
55         ( year % 4 == 0 && year % 100 != 0 ) ) )
56         return testDay;
57
58     cout << "Invalid day (" << testDay << ") set to 1.\n";
59     return 1; // leave object in consistent state if bad value
60 } // end function checkDay
```

Fig. 10.11 | Date class member-function definitions. (Part 3 of 3.)

```
1 // Fig. 10.12: Employee.h
2 // Employee class definition showing composition.
3 // Member functions defined in Employee.cpp.
4 #ifndef EMPLOYEE_H
5 #define EMPLOYEE_H
6
7 #include <string>
8 #include "Date.h" // include Date class definition
9 using namespace std;
10
11 class Employee
12 {
13 public:
14     Employee( const string &, const string &,
15                const Date &, const Date & );
16     void print() const;
17     ~Employee(); // provided to confirm destruction order
18 private:
19     string firstName; // composition: member object
20     string lastName; // composition: member object
21     const Date birthDate; // composition: member object
22     const Date hireDate; // composition: member object
23 }; // end class Employee
24
25 #endif
```

```
1 // Fig. 10.13: Employee.cpp
2 // Employee class member-function definitions.
3 #include <iostream>
4 #include "Employee.h" // Employee class definition
5 #include "Date.h" // Date class definition
6 using namespace std;
7
8 // constructor uses member initializer list to pass initializer
9 // values to constructors of member objects
10 Employee::Employee( const string &first, const string &last,
11                     const Date &dateOfBirth, const Date &dateOfHire )
12     : firstName( first ), // initialize firstName
13       lastName( last ), // initialize lastName
14       birthDate( dateOfBirth ), // initialize birthDate
15       hireDate( dateOfHire ) // initialize hireDate
16 {
17     // output Employee object to show when constructor is called
18     cout << "Employee object constructor: "
19             << firstName << ' ' << lastName << endl;
20 } // end Employee constructor
21
```

Fig. 10.13 | Employee class member-function definitions, including constructor with a member initializer list. (Part I of 2.)

```
22 // print Employee object
23 void Employee::print() const
24 {
25     cout << lastName << ", " << firstName << " Hired: ";
26     hireDate.print();
27     cout << " Birthday: ";
28     birthDate.print();
29     cout << endl;
30 } // end function print
31
32 // output Employee object to show when its destructor is called
33 Employee::~Employee()
34 {
35     cout << "Employee object destructor: "
36         << lastName << ", " << firstName << endl;
37 } // end ~Employee destructor
```

Fig. 10.13 | Employee class member-function definitions, including constructor with a member initializer list. (Part 2 of 2.)

```
1 // Fig. 10.14: fig10_14.cpp
2 // Demonstrating composition--an object with member objects.
3 #include <iostream>
4 #include "Employee.h" // Employee class definition
5 using namespace std;
6
7 int main()
8 {
9     Date birth( 7, 24, 1949 );
10    Date hire( 3, 12, 1988 );
11    Employee manager( "Bob", "Blue", birth, hire );
12
13    cout << endl;
14    manager.print();
15
16    cout << "\nTest Date constructor with invalid values:\n";
17    Date lastDayOff( 14, 35, 1994 ); // invalid month and day
18    cout << endl;
19 } // end main
```

Fig. 10.14 | Demonstrating composition—an object with member objects. (Part I of 2.)

```
Date object constructor for date 7/24/1949  
Date object constructor for date 3/12/1988  
Employee object constructor: Bob Blue _____  
  
Blue, Bob Hired: 3/12/1988 Birthday: 7/24/1949  
  
Test Date constructor with invalid values:  
Invalid month (14) set to 1.  
Invalid day (35) set to 1.  
Date object constructor for date 1/1/1994  
  
Date object destructor for date 1/1/1994  
Employee object destructor: Blue, 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
```

There are actually five constructor calls when an `Employee` is constructed—two calls to the `string` class's constructor (lines 12–13 of Fig. 10.13), two calls to the `Date` class's default copy constructor (lines 14–15 of Fig. 10.13) and the call to the `Employee` class's constructor.

Fig. 10.14 | Demonstrating composition—an object with member objects. (Part 2 of 2.)

10.4 Friend

- Friendship
 - enables a class designer to specify nonmember functions that can access a class' s non-public members
- 2 types
 - Friend function
 - Friend class
- Friendship is granted, not taken.
 - I say “you are my friend”
 - Not vice versa
 - Security reason
- Friend can access public and non-public members

Syntax: Friendship

- Syntax: friend function

friend ReturnType FunctionName (FunctionInput);

- For example

friend void setX (Count &, int);

- Syntax: friend class

friend class ClassName;

```
1 // Fig. 10.15: fig10_15.cpp
2 // Friends can access private members of a class.
3 #include <iostream>
4 using namespace std;
5
6 // Count class definition
7 class Count
8 {
9     friend void setX( Count &, int ); // friend declaration
10 public:
11     // constructor
12     Count()
13         : x( 0 ) // initialize x to 0
14     {
15         // empty body
16     } // end constructor Count
17
18     // output x
19     void print() const
20     {
21         cout << x << endl;
22     } // end function print
```

Fig. 10.15 | Friends can access private members of a class. (Part I of 3.)

```
23 private:  
24     int x; // data member  
25 }; // end class Count  
26  
27 // function setX can modify private data of Count  
28 // because setX is declared as a friend of Count (line 9)  
29 void setX( Count &c, int val )  
30 {  
31     c.x = val; // allowed because setX is a friend of Count  
32 } // end function setX  
33  
34 int main()  
35 {  
36     Count counter; // create Count object  
37  
38     cout << "counter.x after instantiation: "  
39     counter.print();  
40  
41     setX( counter, 8 ); // set x using a friend function  
42     cout << "counter.x after call to setX friend function: "  
43     counter.print();  
44 } // end main
```

Fig. 10.15 | Friends can access private members of a class. (Part 2 of 3.)

```
counter.x after instantiation: 0  
counter.x after call to setX friend function: 8
```

Fig. 10.15 | Friends can access private members of a class. (Part 3 of 3.)

10.6 this pointer

- **this** pointer
 - an implicit argument to each of a class' s non-static member functions.
 - allows those member functions to access the correct object' s data members and other non-static member functions.
- Objects use the this pointer to reference their data members and member functions.

```
1 // Fig. 10.16: fig10_16.cpp
2 // Using the this pointer to refer to object members.
3 #include <iostream>
4 using namespace std;
5
6 class Test
7 {
8 public:
9     Test( int = 0 ); // default constructor
10    void print() const;
11 private:
12     int x;
13 }; // end class Test
14
15 // constructor
16 Test::Test( int value )
17     : x( value ) // initialize x to value
18 {
19     // empty body
20 } // end constructor Test
```

Fig. 10.16 | this pointer implicitly and explicitly accessing an object's members.
(Part 1 of 3.)

```

21
22 // print x using implicit and explicit this pointers;
23 // the parentheses around *this are required
24 void Test::print() const
25 {
26     // implicitly use the this pointer to access the member x
27     cout << "          x = " << x;
28
29     // explicitly use the this pointer and the arrow operator
30     // to access the member x
31     cout << "\n  this->x = " << this->x;
32
33     // explicitly use the dereferenced this pointer and
34     // the dot operator to access the member x
35     cout << "\n(*this).x = " << ( *this ).x << endl;
36 } // end function print
37
38 int main()
39 {
40     Test testObject( 12 ); // instantiate and initialize testObject
41
42     testObject.print();
43 } // end main

```

Fig. 10.16 | +this pointer implicitly and explicitly accessing an object's members

(Part

```

x = 12
this->x = 12
(*this).x = 12

```

Fig. 10.16 | this pointer implicitly and explicitly accessing an object's members.

(Part 3 of 3.)

Example: this pointer

- Use of the this pointer is to enable cascaded member-function calls
- Example:
 - `t.setHour(18).setMinute(30).setSecond(22);`
 - (1) evaluates `t.setHour(18)`, then returns a reference to object t
 - (2) The remaining expression is now
 - `t.setMinute(30).setSecond(22);`
 - (3) The `t.setMinute(30)` call executes and returns a reference to the object t.
 - (4) The remaining expression is interpreted as
 - `t.setSecond(22);`

Time & sethour (int m);
& return type

```
1 // Fig. 10.17: Time.h
2 // Cascading member function calls.
3
4 // Time class definition.
5 // Member functions defined in Time.cpp.
6 #ifndef TIME_H
7 #define TIME_H
8
9 class Time
10 {
11 public:
12     Time( int = 0, int = 0, int = 0 ); // default constructor
13
14     // set functions (the Time & return types enable cascading)
15     Time &setTime( int, int, int ); // set hour, minute, second
16     Time &setHour( int ); // set hour
17     Time &setMinute( int ); // set minute
18     Time &setSecond( int ); // set second
19
20     // get functions (normally declared const)
21     int getHour() const; // return hour
22     int getMinute() const; // return minute
23     int getSecond() const; // return second
```

Fig. 10.17 | Time class definition modified to enable cascaded member-function calls. (Part 1 of 2.)

```
24
25     // print functions (normally declared const)
26     void printUniversal() const; // print universal time
27     void printStandard() const; // print standard time
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. 10.17 | Time class definition modified to enable cascaded member-function calls. (Part 2 of 2.)

```
1 // Fig. 10.18: Time.cpp
2 // Time class member-function definitions.
3 #include <iostream>
4 #include <iomanip>
5 #include "Time.h" // Time class definition
6 using namespace std;
7
8 // constructor function to initialize private data;
9 // calls member function setTime to set variables;
10 // default values are 0 (see class definition)
11 Time::Time( int hr, int min, int sec )
12 {
13     setTime( hr, min, sec );
14 } // end Time constructor
15
16 // set values of hour, minute, and second
17 Time &Time::setTime( int h, int m, int s ) // note Time & return
18 {
19     setHour( h );
20     setMinute( m );
21     setSecond( s );
22     return *this; // enables cascading
23 } // end function setTime
```

Fig. 10.18 | Time class member-function definitions modified to enable cascaded member-function calls. (Part I of 4.)

```
24
25 // set hour value
26 Time &Time::setHour( int h ) // note Time & return
27 {
28     hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
29     return *this; // enables cascading
30 } // end function setHour
31
32 // set minute value
33 Time &Time::setMinute( int m ) // note Time & return
34 {
35     minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
36     return *this; // enables cascading
37 } // end function setMinute
38
39 // set second value
40 Time &Time::setSecond( int s ) // note Time & return
41 {
42     second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
43     return *this; // enables cascading
44 } // end function setSecond
45
```

Fig. 10.18 | Time class member-function definitions modified to enable cascaded member-function calls. (Part 2 of 4.)

```
46 // get hour value
47 int Time::getHour() const
48 {
49     return hour;
50 } // end function getHour
51
52 // get minute value
53 int Time::getMinute() const
54 {
55     return minute;
56 } // end function getMinute
57
58 // get second value
59 int Time::getSecond() const
60 {
61     return second;
62 } // end function getSecond
63
```

Fig. 10.18 | Time class member-function definitions modified to enable cascaded member-function calls. (Part 3 of 4.)

```
64 // print Time in universal-time format (HH:MM:SS)
65 void Time::printUniversal() const
66 {
67     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
68         << setw( 2 ) << minute << ":" << setw( 2 ) << second;
69 } // end function printUniversal
70
71 // print Time in standard-time format (HH:MM:SS AM or PM)
72 void Time::printStandard() const
73 {
74     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
75         << ":" << setfill( '0' ) << setw( 2 ) << minute
76         << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
77 } // end function printStandard
```

Fig. 10.18 | Time class member-function definitions modified to enable cascaded member-function calls. (Part 4 of 4.)

```
1 // Fig. 10.19: fig10_19.cpp
2 // Cascading member-function calls with the this pointer.
3 #include <iostream>
4 #include "Time.h" // Time class definition
5 using namespace std;
6
7 int main()
8 {
9     Time t; // create Time object
10
11    // cascaded function calls
12    t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
13
14    // output time in universal and standard formats
15    cout << "Universal time: ";
16    t.printUniversal();
17
18    cout << "\nStandard time: ";
19    t.printStandard();
20
21    cout << "\n\nNew standard time: ";
22
```

Fig. 10.19 | Cascading member-function calls with the `this` pointer. (Part I of 2.)

```
23     // cascaded function calls  
24     t.setTime( 20, 20, 20 ).printStandard();  
25     cout << endl;  
26 } // end main
```

Universal time: 18:30:22
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM

Fig. 10.19 | Cascading member-function calls with the `this` pointer. (Part 2 of 2.)

10.6 static class member

- Static class member
 - Static data member
 - Static member function
- Class-wide information
 - 1 copy of data is shared by multiple objects in the class
- Syntax ::
 - `ClassName::PublicMemberFunction`
 - `ClassName::PublicDataMember`
- How to access private **static** data member?
 - Use public static member function

```
1 // Fig. 10.20: Employee.h
2 // Employee class definition with a static data member to
3 // track the number of Employee objects in memory
4 #ifndef EMPLOYEE_H
5 #define EMPLOYEE_H
6
7 #include <string>
8 using namespace std;
9
10 class Employee
11 {
12 public:
13     Employee( const string &, const string & ); // constructor
14     ~Employee(); // destructor
15     string getFirstName() const; // return first name
16     string getLastNames() const; // return last name
17
18     // static member function
19     static int getCount(); // return number of objects instantiated
20 private:
21     string firstName;
22     string lastName;
```

Fig. 10.20 | Employee class definition with a `static` data member to track the number of Employee objects in memory. (Part I of 2.)

```
23
24     // static data
25     static int count; // number of objects instantiated
26 }; // end class Employee
27
28 #endif
```

Fig. 10.20 | Employee class definition with a `static` data member to track the number of Employee objects in memory. (Part 2 of 2.)

```
1 // Fig. 10.21: Employee.cpp
2 // Employee class member-function definitions.
3 #include <iostream>
4 #include "Employee.h" // Employee class definition
5 using namespace std;
6
7 // define and initialize static data member at global namespace scope
8 int Employee::count = 0; // cannot include keyword static
9
10 // define static member function that returns number of
11 // Employee objects instantiated (declared static in Employee.h)
12 int Employee::getCount()
13 {
14     return count;
15 } // end static function getCount
16
17 // constructor initializes non-static data members and
18 // increments static data member count
19 Employee::Employee( const string &first, const string &last )
20     : firstName( first ), lastName( last )
21 {
22     ++count; // increment static count of employees
```

Fig. 10.21 | Employee class member-function definitions. (Part I of 2.)

```
23     cout << "Employee constructor for " << firstName
24         << ' ' << lastName << " called." << endl;
25 } // end Employee constructor
26
27 // destructor deallocates dynamically allocated memory
28 Employee::~Employee()
29 {
30     cout << "~Employee() called for " << firstName
31         << ' ' << lastName << endl;
32     --count; // decrement static count of employees
33 } // end ~Employee destructor
34
35 // return first name of employee
36 string Employee::getFirstName() const
37 {
38     return firstName; // return copy of first name
39 } // end function getFirstName
40
41 // return last name of employee
42 string Employee::getLastName() const
43 {
44     return lastName; // return copy of last name
45 } // end function getLastName
```

Fig. 10.21 | Employee class member-function definitions. (Part 2 of 2.)

```
1 // Fig. 10.22: fig10_22.cpp
2 // static data member tracking the number of objects of a class.
3 #include <iostream>
4 #include "Employee.h" // Employee class definition
5 using namespace std;
6
7 int main()
8 {
9     // no objects exist; use class name and binary scope resolution
10    // operator to access static member function getCount
11    cout << "Number of employees before instantiation of any objects is "
12        << Employee::getCount() << endl; // use class name
13
14    // the following scope creates and destroys
15    // Employee objects before main terminates
16    {
17        Employee e1( "Susan", "Baker" );
18        Employee e2( "Robert", "Jones" );
19
20        // two objects exist; call static member function getCount again
21        // using the class name and the binary scope resolution operator
22        cout << "Number of employees after objects are instantiated is "
23            << Employee::getCount();
```

Fig. 10.22 | static data member tracking the number of objects of a class. (Part I)

```
24
25     cout << "\n\nEmployee 1: "
26         << e1.getFirstName() << " " << e1.getLastName()
27         << "\nEmployee 2: "
28         << e2.getFirstName() << " " << e2.getLastName() << "\n\n";
29 } // end nested scope in main
30
31 // no objects exist, so call static member function getCount again
32 // using the class name and the binary scope resolution operator
33 cout << "\nNumber of employees after objects are deleted is "
34     << Employee::getCount() << endl;
35 } // end main
```

Number of employees before instantiation of any objects is 0

Employee constructor for Susan Baker called.

Employee constructor for Robert Jones called.

Number of employees after objects are instantiated is 2

Employee 1: Susan Baker

Employee 2: Robert Jones

~Employee() called for Robert Jones

~Employee() called for Susan Baker

Number of employees after objects are deleted is 0

Fig. 10.22 | static data member tracking the number of objects of a class. (Part 3 of 3.)

10.7 information hiding and data abstraction

- Information Hiding (Encapsulation)
 - Classes hide the details of their implementation from their clients
 - For example
 - Sorting could be implemented in different ways
 - Sorting algorithms
 - Array v.s. Pointer
- Data abstraction
 - The client cares about what functionality a stack offers, not about how that functionality is implemented

Review

- Constant
 - Const object
 - Const member function
- Object as class member
- *Friend*
 - class
 - function
- *this* pointer
- static class member
- Concept: data abstraction and information hiding