

Chapter 7: Classes Part II

1

Outline

- 7.1 Introduction
- 7.2 `const` (Constant) Objects and `const` Member Functions
- 7.3 Composition: Objects as Members of Classes
- 7.4 `friend` Functions and `friend` Classes
- 7.5 Using the `this` Pointer
- 7.6 Dynamic Memory Management with Operators `new` and `delete`
- 7.7 `static` Class Members
- 7.8 Data Abstraction and Information Hiding
 - 7.8.1 Example: Array Abstract Data Type
 - 7.8.2 Example: String Abstract Data Type
 - 7.8.3 Example: Queue Abstract Data Type
- 7.9 Container Classes and Iterators
- 7.10 Proxy Classes



7.1 Introduction

2

- Classes
- Data abstraction
- Object-based programming (OBP)
 - Chapters 6-8
- Inheritance and polymorphism
 - Chapters 9 and 10



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

- Principle of least privilege
 - Only allow modification of necessary objects
- Keyword **const**
 - Specify object not modifiable
 - Compiler error if attempt to modify **const** object
 - Example

```
const Time noon( 12, 0, 0 );
```

 - Declares **const** object **noon** of class **Time**
 - Initializes to 12



7.2 **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



7.2 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



```

1 // Fig. 7.1: time5.h
2 // Definition of class Time.
3 // Member functions defined in time5.cpp.
4 #ifndef TIMES_H
5 #define TIMES_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
23     // print functions (normally declared const)
24     void printUniversal() const; // print universal time
25     void printStandard(); // print standard time

```

Declare **const** get functions.

Declare **const** function
printUniversal.



Outline

time5.h (1 of 2)

```
26
27 private:
28     int hour;    // 0 - 23 (24-hour clock format)
29     int minute; // 0 - 59
30     int second; // 0 - 59
31
32 }; // end class Time
33
34 #endif
```



Outline

7

time5.h (2 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

```
1 // Fig. 7.2: time5.cpp
2 // Member-function definitions for class Time.
3 #include <iostream>
4
5 using std::cout;
6
7 #include <iomanip>
8
9 using std::setfill;
10 using std::setw;
11
12 // include definition of class Time from time5.h
13 #include "time5.h"
14
15 // constructor function to initialize private data;
16 // calls member function setTime to set variables;
17 // default values are 0 (see class definition)
18 Time::Time( int hour, int minute, int second )
19 {
20     setTime( hour, minute, second );
21 }
22 // end Time constructor
23
```



Outline

8

time5.cpp (1 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

24 // set hour, minute and second values
25 void Time::setTime( int hour, int minute, int second )
26 {
27     setHour( hour );
28     setMinute( minute );
29     setSecond( second );
30
31 } // end function setTime
32
33 // set hour value
34 void Time::setHour( int h )
35 {
36     hour = ( h >= 0 && h < 24 ) ? h : 0;
37
38 } // end function setHour
39
40 // set minute value
41 void Time::setMinute( int m )
42 {
43     minute = ( m >= 0 && m < 60 ) ? m : 0;
44
45 } // end function setMinute
46

```



```

47 // set second value
48 void Time::setSecond( int s )
49 {
50     second = ( s >= 0 && s < 60 ) ? s : 0;
51
52 } // end function setSecond
53
54 // return hour value
55 int Time::getHour() const
56 {
57     return hour;
58
59 } // end function getHour
60
61 // return minute value
62 int Time::getMinute() const
63 {
64     return minute;
65
66 } // end function getMinute
67

```



const functions do not
modify objects.

```

68 // return second value
69 int Time::getSecond() const
70 {
71     return second;
72 }
73 // end function getSecond
74
75 // print Time in universal format
76 void Time::printUniversal() const
77 {
78     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
79         << setw( 2 ) << minute << ":"
80         << setw( 2 ) << second;
81 }
82 // end function printUniversal
83
84 // print Time in standard format
85 void Time::printStandard() // note lack of const declaration
86 {
87     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
88         << ":" << setfill( '0' ) << setw( 2 ) << minute
89         << ":" << setw( 2 ) << second
90         << ( hour < 12 ? " AM" : " PM" );
91 }
92 // end function printStandard

```

Outline 11

time5.cpp (4 of 4)

const functions do not modify objects.

© 2003 Prentice Hall, Inc. All rights reserved.

```

1 // Fig. 7.3: fig07_03.cpp
2 // Attempting to access a const object with
3 // non-const member functions.
4
5 // include Time class definition from time5.h
6 #include "time5.h"
7
8 int main()
9 {
10     Time wakeUp( 6, 45, 0 ); // non-constant object
11     const Time noon( 12, 0, 0 ); // constant object
12

```

Outline 12

fig07_03.cpp (1 of 2)

Declare **noon** a const object.

Note that non-const constructor can initialize const object.

© 2003 Prentice Hall, Inc. All rights reserved.

13
14
15
16
17
18
19
20
21
22
23
24
25
26
27

```

wakeUp.setHour( 18 ); // OBJECT      MEMBER FUNCTION
noon.setHour( 12 ); // non-const    non-const
wakeUp.getHour();    // non-const    const
noon.getMinute();    // const      const
noon.printUniversal(); // const      const
noon.printStandard(); // const      non-const
return 0;
} // end main

```

Attempting to invoke non-const member function on const object results in compiler error.

Attempting to invoke non-const member function on const object results in compiler error even if function does not modify object.

d:\cpphtp4_examples\ch07\fig07_01\fig07_01.cpp(16) : error C2662: 'setHour' : cannot convert 'this' pointer from 'const class Time &' to 'class Time &'
Conversion loses qualifiers

d:\cpphtp4_examples\ch07\fig07_01\fig07_01.cpp(23) : error C2662: 'printStandard' : cannot convert 'this' pointer from 'const class Time &' to 'class Time &'
Conversion loses qualifiers

Outline 13

fig07_03.cpp (2 of 2)

fig07_03.cpp output (1 of 1)

© 2003 Prentice Hall, Inc. All rights reserved.

7.2 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



```

1 // Fig. 7.4: fig07_04.cpp
2 // Using a member initializer to initialize a
3 // constant of a built-in data type.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 class Increment {
10
11 public:
12     Increment( int c = 0, int i = 1 ); // default constructor
13
14     void addIncrement()
15     {
16         count += increment;
17
18     } // end function addIncrement
19
20     void print() const; // prints count and increment
21

```



```

22 private:
23     int count;
24     const int increment; // const data member
25
26 }; // end class Increment
27
28 // constructor
29 Increment::Increment(
30     :count( c ), // initialize count
31     increment( i ) // required member initializer
32 {
33     // empty body
34
35 } // end Increment constructor
36
37 // print count and increment values
38 void Increment::print() const
39 {
40     cout << "count = " << count
41         << ", increment = " << increment << endl;
42
43 } // end function print
44

```



Member initializers are separated by a colon.

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 (**0**).


```

45 int main()
46 {
47     Increment value( 10, 5 );
48
49     cout << "Before incrementing: ";
50     value.print();
51
52     for ( int j = 0; j < 3; j++ ) {
53         value.addIncrement();
54         cout << "After increment " << j + 1 << ": ";
55         value.print();
56     }
57
58     return 0;
59
60 } // end main

```

```

Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
After increment 3: count = 25, increment = 5

```



Outline

17

fig07_04.cpp
(3 of 3)

fig07_04.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

1 // Fig. 7.5: fig07_05.cpp
2 // Attempting to initialize a constant of
3 // a built-in data type with an assignment.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 class Increment {
10
11 public:
12     Increment( int c = 0, int i = 1 ); // default constructor
13
14     void addIncrement()
15     {
16         count += increment;
17     }
18 } // end function addIncrement
19
20 void print() const; // prints count and increment
21

```



Outline

18

fig07_05.cpp
(1 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.

19
Outline
fig07_05.cpp
(2 of 3)

```

22 private:
23     int count;
24     const int increment; // const data member
25
26 }; // end class Increment
27
28 // constructor
29 Increment::Increment( int c, int i )
30 {
31     // Constant member
32     count = c; // allowed because count is not const
33     increment = i; // ERROR: Cannot modify a const object
34 } // end Increment constructor
35
36 // print count and increment values
37 void Increment::print() const
38 {
39     cout << "count = " << count
40         << ", increment = " << increment << endl;
41 }
42 // end function print
43

```

Declare increment as const data member

Attempting to modify const data member increment results in error.

© 2003 Prentice Hall, Inc.
 All rights reserved.

20
Outline
fig07_05.cpp
(3 of 3)

fig07_05.cpp
output (1 of 1)

```

44 int main()
45 {
46     Increment value( 10, 5 );
47
48     cout << "Before incrementing: ";
49     value.print();
50
51     for ( int j = 0; j < 3; j++ ) {
52         value.addIncrement();
53         cout << "After increment " << j + 1 << ": ";
54         value.print();
55     }
56
57     return 0;
58 } // end main

```

Not using member initializer syntax to initialize const data member increment results in error.

Attempting to modify const data member increment results in error.

D:\cpphtp4_examples\ch07\Fig07_03\Fig07_03.cpp(30) : error C2166: 'increment' : must be initialized in constructor base/member initializer list
 D:\cpphtp4_examples\ch07\Fig07_03\Fig07_03.cpp(24) : see declaration of 'increment'
 D:\cpphtp4_examples\ch07\Fig07_03\Fig07_03.cpp(32) : error C2166: l-value specifies const object

© 2003 Prentice Hall, Inc.
 All rights reserved.

7.3 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)



```

1 // Fig. 7.6: date1.h
2 // Date class definition.
3 // Member functions defined in date1.cpp
4 #ifndef DATE1_H
5 #define DATE1_H
6
7 class Date {
8
9 public:
10     Date( int = 1, int = 1, int = 1 ) {}
11     void print() const; // print date in YYYY-MM-DD format
12     ~Date(); // provided to confirm destruction order
13
14 private:
15     int month; // 1-12 (January-December)
16     int day; // 1-31 based on month
17     int year; // any year
18
19     // utility function to test proper day for month and year
20     int checkDay( int ) const;
21
22 }; // end class Date
23
24 #endif

```

Outline

22

date1.h (1 of 1)

Note no constructor with parameter of type **Date**. Recall compiler provides default copy constructor.

© 2003 Prentice Hall, Inc. All rights reserved.

```

1 // Fig. 7.7: date1.cpp
2 // Member-function definitions for class Date.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 // include Date class definition from date1.h
9 #include "date1.h"
10
11 // constructor confirms proper value for month; calls
12 // utility function checkDay to confirm proper value for day
13 Date::Date( int mn, int dy, int yr )
14 {
15     if ( mn > 0 && mn <= 12 ) // validate the month
16         month = mn;
17
18     else { // invalid month set to 1
19         month = 1;
20         cout << "Month " << mn << " invalid. Set to month 1.\n";
21     }
22
23     year = yr; // should validate yr
24     day = checkDay( dy ); // validate the day
25

```



```

26 // output Date object to show when its constructor is called
27 cout << "Date object constructor for date ";
28 print();
29 cout << endl;
30
31 } // end Date constructor
32
33 // print Date object in form month/day/year
34 void Date::print() const
35 {
36     cout << month << '/' << day << '/' << year;
37 } // end function print
38
39 // output Date object to show when it is destroyed
40 Date::~Date()
41 {
42     cout << "Date object destructor for date ";
43     print();
44     cout << endl;
45 } // end destructor ~Date
46
47 } // end Date class definition
48

```

No arguments; each member function contains implicit handle to object on which it operates.

Output to show timing of destructors.



```

49 // utility function to confirm proper day value based on
50 // month and year; handles leap years, too
51 int Date::checkDay( int testDay ) const
52 {
53     static const int daysPerMonth[ 13 ] =
54         { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
55
56     // determine whether testDay is valid for specified month
57     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
58         return testDay;
59
60     // February 29 check for leap year
61     if ( month == 2 && testDay == 29 &&
62         ( year % 400 == 0 ||
63           ( year % 4 == 0 && year % 100 != 0 ) ) )
64         return testDay;
65
66     cout << "Day " << testDay << " invalid. Set to day 1.\n";
67
68     return 1; // leave object in consistent state if bad value
69
70 } // end function checkDay

```



```

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.

```
26
27 #endif
```

```
1 // Fig. 7.9: employee1.cpp
2 // Member-function definitions for class Employee.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <cstring> // strcpy and strlen prototypes
9
10 #include "employee1.h" // Employee class definition
11 #include "date1.h" // Date class definition
12
```



employee1.h (2 of 2)

employee1.cpp
(1 of 3)

```
13 // constructor uses member initializer list to pass initializer
14 // values to constructors of member objects birthDate and
15 // hireDate [Note: This invokes the so-called "default copy
16 // constructor" which the C++ compiler provides implicitly.]
17 Employee::Employee( const char *first, const char *last,
18     const Date &dateOfBirth, const Date &dateOfHire )
19     : birthDate( dateOfBirth ), // initialize birthDate
20     hireDate( dateOfHire ) // initialize hireDate
21 {
22     // copy first into firstName and be sure
23     int length = strlen( first );
24     length = ( length < 25 ? length : 24 );
25     strncpy( firstName, first, length );
26     firstName[ length ] = '\0';
27
28     // copy last into lastName and be sure that it fits
29     length = strlen( last );
30     length = ( length < 25 ? length : 24 );
31     strncpy( lastName, last, length );
32     lastName[ length ] = '\0';
33
34     // output Employee object to show when constructor is called
35     cout << "Employee object constructor: "
36         << firstName << " " << lastName << endl;
37
```

Member initializer syntax to initialize Date data members birthDate and hireDate; compiler uses default copy constructor.

Output to show timing of constructors.



employee1.cpp
(2 of 3)

```

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
52 Employee::~Employee()
53 {
54     cout << "Employee object destructor: "
55           << lastName << ", " << firstName << endl;
56
57 } // end destructor -Employee

```

Output to show timing of destructors.



Outline

29

employee1.cpp
(3 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

1 // Fig. 7.10: fig07_10.cpp
2 // Demonstrating composition--an object with member objects.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include "employee1.h" // Employee class definition
9
10 int main()
11 {
12     Date birth( 7, 24, 1949 );
13     Date hire( 3, 12, 1988 );
14     Employee manager( "Bob", "Jones", birth, hire );
15
16     cout << '\n';
17     manager.print();
18
19     cout << "\nTest Date constructor with invalid values:\n";
20     Date lastDayOff( 14, 35, 1994 ); // invalid month and day
21     cout << endl;
22
23     return 0;
24
25 } // end main

```

Create Date objects to pass to Employee constructor.



Outline

30

fig07_10.cpp
(1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

31

Outline

10.cpp
t (1 of 1)

```

Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Jones

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

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

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

```

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

Destructor for **Employee's**

Destructor for **Employee's**

Destructor for **Date** object **birth.**

© 2003 Prentice Hall, Inc.
All rights reserved.

32

7.4 friend Functions and friend Classes

- **friend** function
 - Defined outside class's scope
 - Right to access non-public members
- Declaring **friends**
 - Function
 - Precede function prototype with keyword **friend**
 - All member functions of class **ClassTwo** as **friends** of class **ClassOne**
 - Place declaration of form


```

friend class ClassTwo;
in ClassOne definition
              
```

© 2003 Prentice Hall, Inc. All rights reserved.

7.4 friend Functions and friend Classes

- Properties of friendship
 - Friendship granted, not taken
 - Class **B friend** of class **A**
 - Class **A** must explicitly declare class **B friend**
 - Not symmetric
 - Class **B friend** of class **A**
 - Class **A** not necessarily **friend** of class **B**
 - Not transitive
 - Class **A friend** of class **B**
 - Class **B friend** of class **C**
 - Class **A** not necessarily **friend** of Class **C**



```

1 // Fig. 7.11: fig07_11.cpp
2 // Friends can access private members of a class.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 // Count class definition
9 class Count {
10     friend void setX( Count &, int ); // friend declaration
11
12 public:
13
14     // constructor
15     Count()
16         : x( 0 ) // initialize x to 0
17     {
18         // empty body
19     } // end Count constructor
20
21

```

Outline 34

fig07_11.cpp
(1 of 3)

Precede function prototype with keyword **friend**.

© 2003 Prentice Hall, Inc.
All rights reserved.

```

22 // output x
23 void print() const
24 {
25     cout << x << endl;
26
27 } // end function print
28
29 private:
30     int x; // data member
31
32 }; // end class Count
33
34 // function setX can
35 // because setX is de
36 void setX( Count &c,
37 {
38     c.x = val; // leg
39
40 } // end function setX
41

```

Since **setX** friend of **Count**, can access and modify **private** data member **x**.

Pass **Count** object since C++ is a **reference** function.



```

42 int main()
43 {
44     Count counter; // create Count object
45
46     cout << "counter.x after instantiation: ";
47     counter.print();
48
49     setX( counter, 8 ); // set x with a friend
50
51     cout << "counter.x after call to setX friend function: ";
52     counter.print();
53
54     return 0;
55
56 } // end main

```

Use **friend** function to access and modify **private** data member **x**.

```

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

```



```

1 // Fig. 7.12: fig07_12.cpp
2 // Non-friend/non-member functions cannot access
3 // private data of a class.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 // Count class definition
10 // (note that there is no friendship declaration)
11 class Count {
12
13 public:
14
15     // constructor
16     Count()
17         : x( 0 ) // initialize x to 0
18     {
19         // empty body
20
21     } // end Count constructor
22

```



```

23 // output x
24 void print() const
25 {
26     cout << x << endl;
27
28 } // end function print
29
30 private:
31     int x; // data member
32
33 }; // end class Count
34
35 // function tries to modify
36 // but cannot because function
37 void cannotSetX( Count &c, int i)
38 {
39     c.x = val; // ERROR: cannot access private member in Count
40
41 } // end function cannotSetX
42

```

Attempting to modify
private data member from
non-friend function results
in error.



```
43 int main()
44 {
45     Count counter;           // create Count object
46
47     cannotSetX( counter, 3 ); // cannotSetX is not a friend
48
49     return 0;
50
51 } // end main
```

D:\cpphtp4_examples\ch07\Fig07_12\Fig07_12.cpp(39) : error C2248:
'x' : cannot access private member declared in class 'Count'
D:\cpphtp4_examples\ch07\Fig07_12\Fig07_12.cpp(31) :
see declaration of 'x'

Attempting to modify
private data member from
non-**friend** function results
in error.

39

Outline

fig07_12.cpp
(3 of 3)

fig07_12.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

7.5 Using the **this** Pointer

- **this** pointer
 - Allows object to access own address
 - Not part of object itself
 - Implicit argument to non-**static** member function call
 - Implicitly reference member data and functions
 - Type of **this** pointer depends on
 - Type of object
 - Whether member function is **const**
 - In non-**const** member function of **Employee**
 - **this** has type **Employee * const**
 - Constant pointer to non-constant **Employee** object
 - In **const** member function of **Employee**
 - **this** has type **const Employee * const**
 - Constant pointer to constant **Employee** object

© 2003 Prentice Hall, Inc. All rights reserved.

```

1 // Fig. 7.13: fig07_13.cpp
2 // Using the this pointer to refer to object members.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 class Test {
9
10 public:
11     Test( int = 0 );    // default constructor
12     void print() const;
13
14 private:
15     int x;
16
17 }; // end class Test
18
19 // constructor
20 Test::Test( int value )
21     : x( value ) // initialize x to value
22 {
23     // empty body
24
25 } // end Test constructor

```



```

26
27 // print x using implicit and explicit this pointer
28 // parentheses around *this required
29 void Test::print() const
30 {
31     // implicitly use this pointer to access member
32     cout << "    x = " << x;
33
34     // explicitly use this pointer to access member
35     cout << "\n this->x = " << this->x;
36
37     // explicitly use dereferenced this pointer and
38     // the dot operator to access member x
39     cout << "\n(*this).x = " << ( *this ).x << endl;
40
41 } // end function print
42
43 int main()
44 {
45     Test testObject( 12 );
46
47     testObject.print();
48
49     return 0;
50

```



Implicitly use **this** pointer;
only specify name of data member

Explicitly use **this** pointer
with arrow operator.

Explicitly use **this** pointer;
dereference **this** pointer
first, then use dot operator.

```
51 } // end main
```

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



Outline

43

fig07_13.cpp
(3 of 3)

fig07_13.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

44

7.5 Using the `this` Pointer

- Cascaded member function calls
 - Multiple functions invoked in same statement
 - Function returns reference pointer to same object

```
{ return *this; }
```
 - Other functions operate on that pointer
 - Functions that do not return references must be called last



```

1 // Fig. 7.14: time6.h
2 // Cascading member function calls.
3
4 // Time class definition.
5 // Member functions defined in time6.cpp.
6 #ifndef TIME6_H
7 #define TIME6_H
8
9 class Time {
10
11 public:
12     Time( int = 0, int = 0, int = 0 );
13
14     // set functions
15     Time &setTime( int, int, int ); // set time
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
24

```

Set functions return reference to **Time** object to enable cascaded member function calls.



```

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

```



```

1 // Fig. 7.15: time6.cpp
2 // Member-function definitions for Time class.
3 #include <iostream>
4
5 using std::cout;
6
7 #include <iomanip>
8
9 using std::setfill;
10 using std::setw;
11
12 #include "time6.h" // Time class definition
13
14 // constructor function to initialize private data;
15 // calls member function setTime to set variables;
16 // default values are 0 (see class definition)
17 Time::Time( int hr, int min, int sec )
18 {
19     setTime( hr, min, sec );
20
21 } // end Time constructor
22

```



```

23 // set values of hour, minute, and second
24 Time &Time::setTime( int h, int m, int s )
25 {
26     setHour( h );
27     setMinute( m );
28     setSecond( s );
29
30     return *this; ▲ // enables cascading
31
32 } // end function setTime
33
34 // set hour value
35 Time &Time::setHour( int h )
36 {
37     hour = ( h >= 0 && h < 24 ) ? h : 0;
38
39     return *this; ▲ // enables cascading
40
41 } // end function setHour
42

```

Return *this as reference to enable cascaded member function calls.

Return *this as reference to enable cascaded member function calls.




```
43 // set minute value
44 Time &Time::setMinute( int m )
45 {
46     minute = ( m >= 0 && m < 60 )
47
48     return *this; // enables cascading
49
50 } // end function setMinute
51
52 // set second value
53 Time &Time::setSecond( int s )
54 {
55     second = ( s >= 0 && s < 60 )
56
57     return *this; // enables cascading
58
59 } // end function setSecond
60
61 // get hour value
62 int Time::getHour() const
63 {
64     return hour;
65
66 } // end function getHour
67
```

Return ***this** as reference to enable cascaded member function calls.

Return ***this** as reference to enable cascaded member function calls.



```
68 // get minute value
69 int Time::getMinute() const
70 {
71     return minute;
72
73 } // end function getMinute
74
75 // get second value
76 int Time::getSecond() const
77 {
78     return second;
79
80 } // end function getSecond
81
82 // print Time in universal format
83 void Time::printUniversal() const
84 {
85     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
86           << setw( 2 ) << minute << ":"
87           << setw( 2 ) << second;
88
89 } // end function printUniversal
90
```



```
91 // print Time in standard format
92 void Time::printStandard() const
93 {
94     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
95     << ":" << setfill( '0' ) << setw( 2 ) << minute
96     << ":" << setw( 2 ) << second
97     << ( hour < 12 ? " AM" : " PM" );
98
99 } // end function printStandard
```



```
1 // Fig. 7.16: fig07_16.cpp
2 // Cascading member function calls with the this pointer.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include "time6.h" // Time class definition
9
10 int main()
11 {
12     Time t;
13
14     // cascaded function calls
15     t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
16
17     // output time in universal and standard formats
18     cout << "Universal time: ";
19     t.printUniversal();
20
21     cout << "\nStandard time: ";
22     t.printStandard();
23
24     cout << "\n\nNew standard time: ";
25
```

Cascade member function calls; recall dot operator associates from left to right.



```
26 // cascaded function calls
27 t.setTime( 20, 20, 20 ).printStandard();
28
29 cout << endl;
30
31 return 0;
32
33 } // end main
```

Universal time: 18:30:22
Standard time: 6:30:22 PM
New standard time: 8:20:20 PM

Function call to `printStandard` must appear last; `printStandard` does not return reference to `t`.

Outline 53

_16.cpp 2)
_16.cpp out (1 of 1)

© 2003 Prentice Hall, Inc. All rights reserved.

7.6 Dynamic Memory Management with Operators `new` and `delete`

- Dynamic memory management
 - Control allocation and deallocation of memory
 - Operators `new` and `delete`
 - Include standard header `<new>`
 - Access to standard version of `new`



7.6 Dynamic Memory Management with Operators `new` and `delete`

- **new**

- Consider

```
Time *timePtr;
timePtr = new Time;
```

- **new** operator

- Creates object of proper size for type `Time`
 - Error if no space in memory for object
- Calls default constructor for object
- Returns pointer of specified type

- Providing initializers

```
double *ptr = new double( 3.14159 );
Time *timePtr = new Time( 12, 0, 0 );
```

- Allocating arrays

```
int *gradesArray = new int[ 10 ];
```



7.6 Dynamic Memory Management with Operators `new` and `delete`

- **delete**

- Destroy dynamically allocated object and free space

- Consider

```
delete timePtr;
```

- Operator **delete**

- Calls destructor for object
- Deallocates memory associated with object
 - Memory can be reused to allocate other objects

- Deallocating arrays

```
delete [] gradesArray;
```

- Deallocates array to which `gradesArray` points

- If pointer to array of objects
 - First calls destructor for each object in array
 - Then deallocates memory



7.7 static Class Members

- **static** class variable
 - “Class-wide” data
 - Property of class, not specific object of class
 - Efficient when single copy of data is enough
 - Only the **static** variable has to be updated
 - May seem like global variables, but have class scope
 - Only accessible to objects of same class
 - Initialized exactly once at file scope
 - Exist even if no objects of class exist
 - Can be **public**, **private** or **protected**



7.7 static Class Members

- Accessing **static** class variables
 - Accessible through any object of class
 - **public static** variables
 - Can also be accessed using binary scope resolution operator (::)


```
Employee::count
```
 - **private static** variables
 - When no class member objects exist
 - Can only be accessed via **public static** member function
 - To call **public static** member function combine class name, binary scope resolution operator (::) and function name


```
Employee::getCount()
```



7.7 static Class Members

- **static** member functions
 - Cannot access non-**static** data or functions
 - No **this** pointer for **static** functions
 - **static** data members and **static** member functions exist independent of objects



```

1 // Fig. 7.17: employee2.h
2 // Employee class definition.
3 #ifndef EMPLOYEE2_H
4 #define EMPLOYEE2_H
5
6 class Employee {
7
8 public:
9     Employee( const char *, const char * ); // constructor
10    ~Employee(); // destructor
11    const char *getFirstName() const; // r
12    const char *getLastName() const; // r
13
14    // static member function
15    static int getCount(); // return # obj
16
17 private:
18    char *firstName;
19    char *lastName;
20
21    // static data member
22    static int count; // number of objects instantiated
23
24 }; // end class Employee
25

```

Outline

60

employee2.h (1 of 2)

static member function can only access **static** data members and member functions.

static data member is class-wide data.

© 2003 Prentice Hall, Inc.
All rights reserved.

```

26 #endif

1 // Fig. 7.18: employee2.cpp
2 // Member-function definitions for class Employee.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <new> // C++ standard new operator
9 #include <cstring> // strcpy and strlen prototypes
10
11 #include "employee2.h" // Employee class
12
13 // define and initialize static data member
14 int Employee::count = 0;
15
16 // define static member function that returns
17 // Employee objects instantiated
18 int Employee::getCount()
19 {
20     return count;
21 }
22 // end static function getCount

```

61

Outline

employee2.h (2 of 2)

employee2.cpp (1 of 3)

Initialize **static** data member exactly once at file scope.

static member function accesses **static** data member **count**.

© 2003 Prentice Hall, Inc. All rights reserved.

```

23
24 // constructor dynamically allocates space for
25 // first and last name and uses strcpy to copy
26 // first and last names into the object
27 Employee::Employee( const char *first, const char *last )
28 {
29     firstName = new char[ strlen( first ) + 1 ];
30     strcpy( firstName, first );
31
32     lastName = new char[ strlen( last ) + 1 ];
33     strcpy( lastName, last );
34     ++count; // increment static count of employees
35
36     cout << "Employee constructor for " << firstName
37         << " " << lastName << " called." << endl;
38
39 } // end Employee constructor
40
41
42 // destructor deallocates dynamically allocated memory
43 Employee::~Employee()
44 {
45     cout << "--Employee() called for " << firstName
46         << " " << lastName << endl;
47

```

62

Outline

employee2.cpp

new operator dynamically allocates space.

Use **static** data member to store total **count** of employees.

© 2003 Prentice Hall, Inc. All rights reserved.

```
48 delete [] firstName; // recapture memory
49 delete [] lastName; // recapture memory
50
51 --count; // decrement static count of employees
52
53 } // end destructor -Emp
54
55 // return first name of employee
56 const char *Employee::getFirstName() const
57 {
58     // const before return type prevents client from modifying
59     // private data; client should copy returned string before
60     // destructor deletes storage to prevent undefined pointer
61     return firstName;
62 }
63 // end function getFirstName
64
65 // return last name of employee
66 const char *Employee::getLastName() const
67 {
68     // const before return type prevents client from modifying
69     // private data; client should copy returned string before
70     // destructor deletes storage to prevent undefined pointer
71     return lastName;
72 }
73 // end function getLastName
```

Use static data member to store total count of employees.

allocates



```
1 // Fig. 7.19: fig07_19.cpp
2 // Driver to test class Employee.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <new> // C++ standard new operator
9
10 #include "employee2.h" // Employee class definition
11
12 int main()
13 {
14     cout << "Number of employees before instantiation is "
15          << Employee::getCount() << endl; // use class name
16
17     Employee *e1Ptr = new Employee(1);
18     Employee *e2Ptr = new Employee(2);
19
20     cout << "Number of employees after instantiation is "
21          << e1Ptr->getCount();
22 }
```

static member function can be invoked on any object of class

new operator dynamically creates space.




```
23 cout << "\n\nEmployee 1: "  
24 << e1Ptr->getFirstName()  
25 << " " << e1Ptr->getLastName()  
26 << "\n\nEmployee 2: "  
27 << e2Ptr->getFirstName()  
28 << " " << e2Ptr->getLastName() << "\n\n";  
29  
30 delete e1Ptr; // recapture memory  
31 e1Ptr = 0; // disconnect pointer from free-store space  
32 delete e2Ptr; // recapture memory  
33 e2Ptr = 0; // disconnect pointer f  
34  
35 cout << "Number of employees a  
36 << Employee::getCount() <  
37  
38 return 0;  
39  
40 } // end main
```

Operator
memory
static member function
invoked using binary scope
resolution operator (no
existing class objects).



```
Number of employees before instantiation is 0  
Employee constructor for Susan Baker called.  
Employee constructor for Robert Jones called.  
Number of employees after instantiation is 2  
  
Employee 1: Susan Baker  
Employee 2: Robert Jones  
  
~Employee() called for Susan Baker  
~Employee() called for Robert Jones  
Number of employees after deletion is 0
```



7.8 Data Abstraction and Information Hiding

- Information hiding
 - Classes hide implementation details from clients
 - Example: stack data structure
 - Data elements added (pushed) onto top
 - Data elements removed (popped) from top
 - Last-in, first-out (LIFO) data structure
 - Client only wants LIFO data structure
 - Does not care how stack implemented
- Data abstraction
 - Describe functionality of class independent of implementation



7.8 Data Abstraction and Information Hiding

- Abstract data types (ADTs)
 - Approximations/models of real-world concepts and behaviors
 - `int`, `float` are models for a numbers
 - Data representation
 - Operations allowed on those data
- C++ extensible
 - Standard data types cannot be changed, but new data types can be created



7.8.1 Example: Array Abstract Data Type

- ADT array
 - Could include
 - Subscript range checking
 - Arbitrary range of subscripts
 - Instead of having to start with 0
 - Array assignment
 - Array comparison
 - Array input/output
 - Arrays that know their sizes
 - Arrays that expand dynamically to accommodate more elements



7.8.2 Example: String Abstract Data Type

- Strings in C++
 - C++ does not provide built-in string data type
 - Maximizes performance
 - Provides mechanisms for creating and implementing string abstract data type
 - String ADT (Chapter 8)
 - ANSI/ISO standard **string** class (Chapter 19)



7.8.3 Example: Queue Abstract Data Type

- Queue
 - FIFO
 - First in, first out
 - Enqueue
 - Put items in queue one at a time
 - Dequeue
 - Remove items from queue one at a time
- Queue ADT
 - Implementation hidden from clients
 - Clients may not manipulate data structure directly
 - Only queue member functions can access internal data
 - Queue ADT (Chapter 15)
 - Standard library `queue` class (Chapter 20)



7.9 Container Classes and Iterators

- Container classes (collection classes)
 - Designed to hold collections of objects
 - Common services
 - Insertion, deletion, searching, sorting, or testing an item
 - Examples
 - Arrays, stacks, queues, trees and linked lists
- Iterator objects (iterators)
 - Returns next item of collection
 - Or performs some action on next item
 - Can have several iterators per container
 - Book with multiple bookmarks
 - Each iterator maintains own “position”
 - Discussed further in Chapter 20



7.10 Proxy Classes

- Proxy class
 - Hide implementation details of another class
 - Knows only **public** interface of class being hidden
 - Enables clients to use class's services without giving access to class's implementation
- Forward class declaration
 - Used when class definition only uses pointer to another class
 - Prevents need for including header file
 - Declares class before referencing
 - Format:


```
class ClassToLoad;
```



```

1 // Fig. 7.20: implementation.h
2 // Header file for class Implementation
3
4 class Implementation {
5
6 public:
7
8     // constructor
9     Implementation( int v )
10        : value( v ) // initialize value with v
11    {
12        // empty body
13
14    } // end Implementation constructor
15
16    // set value to v
17    void setValue( int v ) ▲
18    {
19        value = v; // should validate v
20
21    } // end function setValue
22

```



Outline

74

implementation.h
(1 of 2)

public member function.

© 2003 Prentice Hall, Inc.
All rights reserved.

```
23 // return value
24 int getValue() const
25 {
26     return value;
27 }
28 } // end function getValue
29
30 private:
31     int value;
32
33 }; // end class Implementation
```

public member function.



```
1 // Fig. 7.21: interface.h
2 // Header file for interface.cpp
3
4 class Implementation; // forward class declaration
5
6 class Interface {
7
8 public:
9     Interface( int );
10    void setValue( int ); // same public interface
11    int getValue() const; // class Implementation
12    ~Interface();
13
14 private:
15
16 // requires previous forward declaration
17    Implementation *ptr;
18
19 }; // end class Interface
```

Provide same public interface as class Implementation; recall setValue and getValue only public member functions.

Pointer to Implementation object requires forward class declaration.



```

1 // Fig. 7.22: interface.cpp
2 // Definition of class Interface
3 #include "interface.h" // Interface class definition
4 #include "implementation.h" // Implementation definition
5
6 // constructor
7 Interface::Interface( int v )
8 : ptr ( new Implementation( v ) ) // includes header file for class
9 { // Implementation.
10 // empty body
11
12 } // end Interface constructor
13
14 // call Implementation's setValue function
15 void Interface::setValue( int v )
16 {
17     ptr->setValue( v );
18 }
19 // end function setValue
20

```

Maintain pointer to underlying Implementation object.

Invoke corresponding function on underlying Implementation object.



```

21 // call Implementation's getValue function
22 int Interface::getValue() const
23 {
24     return ptr->getValue();
25 }
26 // end function getValue
27
28 // destructor
29 Interface::~Interface()
30 {
31     delete ptr;
32 }
33 // end destructor ~Interface

```

Invoke corresponding function on underlying Implementation object.

Deallocate underlying Implementation object.



```
1 // Fig. 7.23: fig07_23.cpp
2 // Hiding a class's private data with a proxy class.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include "interface.h" // Interface class
9
10 int main()
11 {
12     Interface i( 5 );
13
14     cout << "Interface contains: " << i.getValue()
15          << " before setValue" << endl;
16
17     i.setValue( 10 );
18
19     cout << "Interface contains: " << i.getValue()
20          << " after setValue" << endl;
21
22     return 0;
23
24 } // end main
```

Only include proxy class header file.

Create object of proxy class **Interface**; note no mention of **Implementation** class.

Invoke member functions via proxy class object.

```
Interface contains: 5 before setValue
Interface contains: 10 after setValue
```



fig07_23.cpp
(1 of 1)

fig07_23.cpp
output (1 of 1)