

# 1 Classes Part II

## 1.1 `const` (Constant) Objects and `const` Member Functions

Some objects need to be modifiable and some do not. The programmer may use keyword `const` to specify that an object is not modifiable and that any attempt to modify the object should result in a compiler error.

- 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
- `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
- Constructors and destructors
  - Cannot be `const`
  - Must be able to modify objects
    - \* Constructor; Initializes objects
    - \* Destructor; Performs termination housekeeping

The program of Figs. 1-4 modifies class `Time` by making its *get* functions and `printUniversal` function `const`.

- 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

Figs. 4-6 introduces using *member initializer syntax*. Figs. 7-8 illustrates the compiler errors for a program that attempts to initialize **const** data member **increment** with an assignment statement in the **Increment** constructor's body rather than with a member initializer.

```
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
```



Outline

6

time5.h (1 of 2)

Declare const get functions.

Declare const function printUniversal.

© 2003 Prentice Hall, Inc.  
All rights reserved.

```
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.

Figure 1: **Time** class definition with **const** member functions.

```
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
```



[Outline](#)

9

time5.cpp (2 of 4)

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 2: **Time** class member-function definitions, including **const** member functions. (part 1 of 2)

```

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

```



[Outline](#)

10

time5.cpp (3 of 4)

const functions do not modify objects.

© 2003 Prentice Hall, Inc.  
All rights reserved.

```

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

```



[Outline](#)

11

time5.cpp (4 of 4)

const functions do not modify objects.

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 3: **Time** class member-function definitions, including **const** member functions. (part 2 of 2)

```

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         // OBJECT      MEMBER FUNCTION
14     wakeUp.setHour( 18 ); // non-const non-const
15
16     noon.setHour( 12 ); // const non-const
17
18     wakeUp.getHour(); // non-const const
19
20     noon.getMinute(); // const const
21     noon.printUniversal(); // const const
22
23     noon.printStandard(); // const non-const
24
25     return 0;
26
27 } // end main

```



Outline

13

fig07\_03.cpp  
(2 of 2)

fig07\_03.cpp  
output (1 of 1)

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)
'setHour' : cannot convert 'this' pointer from
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

```

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 4: **const** objects and **const** member functions.

```

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

```



[Outline](#)

15

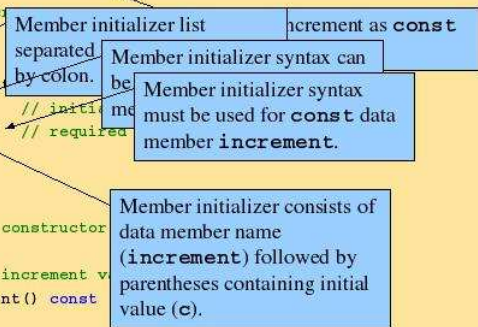
fig07\_04.cpp  
(1 of 3)

© 2003 Prentice Hall, Inc.  
All rights reserved.

```

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
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

```



[Outline](#)

16



fig07\_04.cpp  
(2 of 3)

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 5: Member initializer used to initialize a constant of a built-in data type. (part 1 of 2)

```
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.

Figure 6: Member initializer used to initialize a constant of a built-in data type. (part 2 of 2)



```

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.

```

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.



[Outline](#)

19

fig07\_05.cpp  
(2 of 3)

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 7: Erroneous attempt to initialize a constant of a built-in data type by assignment. (part 1 of 2)

```

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
59 } // end main

```

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

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

Attempting to modify **const**
data member **increment**
results in error.

Outline 20
fig07\_05.cpp
(3 of 3)
fig07\_05.cpp
output (1 of 1)

Figure 8: Erroneous attempt to initialize a constant of a built-in data type by assignment. (part 2 of 2)

```

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
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

```

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

Figure 9: **Date** class definition.

## 1.2 Composition: Objects as Members of Classes

An **AlarmClock** object needs to know when it is supposed to sound its alarm, so why not include a **Time** object as a member of the **AlarmClock** class? Such a capability is called *composition*.

- 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)

The program of Figs. 9-14 uses class **Date** and class **Employee** to demonstrate objects as members of other objects. The colon (:) in the header separates the member initializers from the parameter list. In Fig. 14, when each of the **Employee**’s **Date** member object’s initialized in the **Employee** constructor’s member initializer list, the default copy constructor for class **Date** is called. This constructor is defined implicitly by the compiler and does not contain any output statements.

```

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

```



Outline

23

date1.cpp (1 of 3)

© 2003 Prentice Hall, Inc.  
All rights reserved.

```

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
38 } // end function print
39
40 // output Date object to show when it is destroyed
41 Date::~Date()
42 {
43     cout << "Date object destructor for date ";
44     print();
45     cout << endl;
46
47 } // end destructor ~Date
48

```

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

Output to show timing of destructors.



Outline

24

date1.cpp (2 of 3)

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 10: **Date** class member-function definitions. (part 1 of 2)

```

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

```



[Outline](#)

25

date1.cpp (3 of 3)

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 11: **Date** class member-function definitions. (part 2 of 2)

```
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
```

▲  
▼  
Outline 26  
employee1.h (1 of 2)

Using composition;  
Employee object contains  
Date objects as data  
members.

© 2003 Prentice Hall, Inc.  
All rights reserved.

```
26
27 #endif
```

▲  
▼  
Outline 27  
employee1.h (2 of 2)

```
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.cpp  
(1 of 3)

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 12: Employee class definition showing composition.

```

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.

```

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.

Figure 13: Employee class member-function definitions, including constructor with a member-initializer list.

```

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

```



Outline

30

fig07\_10.cpp  
(1 of 1)

Create Date objects to pass to Employee constructor.

© 2003 Prentice Hall, Inc.  
All rights reserved.

```

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

```



Outline

31

fig07\_10.cpp  
(1 of 1)

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

Destructor for Employee's  
me  
de  
ob  
bi  
Destructor for Employee's  
de  
ob  
bi  
Destructor for Date object  
birth.

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 14: Member-object initializers.



### 1.3 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
- 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**

The program of Figs. 15-16 (top) defines friend function **setX** to set the **private** data member **x** of class **Count**. Friend declaration can appear anywhere in the class. The program of Figs. 16 (bottom) -17 demonstrates the error messages produced by the compiler when nonfriend function **cannot-SetX** is called to modify **private** data member **x**.

```

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

```

Precede function prototype with keyword friend.



Outline

34

fig07\_11.cpp  
(1 of 3)

© 2003 Prentice Hall, Inc.  
All rights reserved.

```

22 // output x
23 void print() const
24 {
25     cout << x << endl;
26 } // end function print
27
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

```

Pass Count object since C-style standalone function.

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



Outline

35

fig07\_11.cpp  
(2 of 3)

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 15: Friends can access **private** members of the class.

```

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

```

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

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



Outline

36

fig07\_11.cpp  
(3 of 3)

fig07\_11.cpp  
output (1 of 1)

© 2003 Prentice Hall, Inc.  
All rights reserved.

```

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     } // end Count constructor
21
22

```



Outline

37

fig07\_12.cpp  
(1 of 3)

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 16: Nonfriend/nonmember functions cannot access **private** members.  
(part 1 of 2)

```

23 // output x
24 void print() const
25 {
26     cout << x << endl;
27 } // end function print
28
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.



Outline

38

fig07\_12.cpp  
(2 of 3)

© 2003 Prentice Hall, Inc.  
All rights reserved.

```

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:\cpphttp4_examples\ch07\Fig07_12\Fig07_12.cpp(39) : error C2248:
'x' : cannot access private member declared in class 'Count'
D:\cpphttp4_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.



Outline

39

fig07\_12.cpp  
(3 of 3)

fig07\_12.cpp  
output (1 of 1)

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 17: Nonfriend/nonmember functions cannot access **private** members. (part 2 of 2)

## 1.4 Using the **this** Pointer

We have seen that an object's member functions can manipulate the object's data. How do member functions know which object's data members to manipulate? Every object has access to its own address through a pointer called **this** (a C++ keyword).

- 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

The program of Figs. 18-19 demonstrates the implicit and explicit use of the **this** pointer to enable a member function of class **Test** to print the **private** data **x** of a **Test** object. The program of Figs. 20-24 modifies class **Time**'s *set* functions **setTime**, **setHour**, **setMinute** and **setSecond** such that each returns a reference to a **Time** object to enable cascaded member-function calls.

- 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. 7.13: 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

```

41

Outline

fig07\_13.cpp  
(1 of 3)

© 2003 Prentice Hall, Inc.  
All rights reserved.

```

26
27 // print x using implicit and explicit this pointers;
28 // parentheses around *this required
29 void Test::print() const
30 {
31     // implicitly use this pointer to access member x
32     cout << "    x = " << x;
33
34     // explicitly use this pointer to access member x
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 (*x*).
- Explicitly use **this** pointer with arrow operator.
- Explicitly use **this** pointer; dereference **this** pointer first, then use dot operator.

42

Outline



fig07\_13.cpp  
(2 of 3)

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 18: **this** pointer implicitly and explicitly used to access an object's members. (part 1 of 2)

```
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.

Figure 19: **this** pointer implicitly and explicitly used to access an object's members. (part 2 of 2)

```

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 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
24

```

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



[Outline](#)

45

time6.h (1 of 2)

© 2003 Prentice Hall, Inc.  
All rights reserved.

```

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

```



[Outline](#)

46

time6.h (2 of 2)

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 20: **Time** class definition modified to enable cascaded member-function calls.



```

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

```



[Outline](#)

47

time6.cpp (1 of 5)

© 2003 Prentice Hall, Inc.  
All rights reserved.

```

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 } // end function setTime
32
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 } // end function setHour
41
42

```

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

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



[Outline](#)

48

time6.cpp (2 of 5)

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 21: **Time** class member-function definitions modified to enable cascaded member-function calls. (part 1 of 3)

```

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.



[Outline](#)

49

time6.cpp (3 of 5)

© 2003 Prentice Hall, Inc.  
All rights reserved.

```

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

```



[Outline](#)

50

time6.cpp (4 of 5)

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 22: **Time** class member-function definitions modified to enable cascaded member-function calls. (part 2 of 3)

```
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
```



Outline

51

time6.cpp (5 of 5)

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 23: **Time** class member-function definitions modified to enable cascaded member-function calls. (part 3 of 3)

```

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.

52

Outline

fig07\_16.cpp (1 of 2)

© 2003 Prentice Hall, Inc.  
All rights reserved.

```

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**.

53

Outline

fig07\_16.cpp (1 of 1)

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 24: Cascading member-function calls.