

A Deeper Look





Deeper into C++ Classes

- const objects and const member functions
- Composition
- Friendship
- · this pointer
- Dynamic memory management
 - new and del ete operators
- static class members and member functions
- . Abstract Data Types



21.2 const (Constant) Objects and const Member Functions

- · Principle of least privilege
 - One of the most fundamental principles of good software engineering
 - Applies to objects, too
- const objects
 - Keyword const
 - Specifies that an object is not modifiable.
 - Attempts to modify the object will result in compilation errors.

Example

- Const Time noon (12, 0, 0);



const (Constant) Objects and const Member Functions

- const member functions
 - Only const member function can be called for const objects.
 - Member functions declared const are not allowed to modify the object.
 - A function is specified as **const** both in its prototype and in its definition.
 - const declarations are not allowed for constructors and destructors.



Software Engineering Observation 21.2

 A const member function can be overloaded with a non-const version. The compiler chooses which overloaded member function to use based on the object on which the function is invoked. If the object is const, the compiler uses the const version. If the object is not const, the compiler uses the non-const version.



1 // Fig. 21.1: Time.h // Definition of class Time. 2 // Member functions defined in Time.cpp. 3 #ifndef TIME H 4 #define TIME H 5 6 class Time 7 8 { public: 9 Time(int = 0, int = 0, int = 0); // default constructor 10 11 // set functions 12 void setTime(int, int, int); // set time 13 void setHour(int); // set hour 14 void setMinute(int); // set minute 15 void setSecond(int); // set second 16 17 // get functions (normally declared const) 18 19 int getHour() const; // return hour int getMinute() const; // return minute 20 int getSecond() const; // return second 21



```
22
23
     // print functions (normally declared const)
24
     void printUniversal() const; // print universal time
25
     void printStandard(); // print standard time (should be const)
26 private:
     int hour; // 0 - 23 (24-hour clock format)
27
    int minute; // 0 - 59
28
29 int second: // 0 - 59
30 }; // end class Time
31
32 #endif
```



```
1 // Fig. 21.2: Time.cpp
  // Member-function definitions for class Time.
2
   #include <iostream>
3
   usi ng std::cout;
4
5
   #include <iomanip>
6
   using std::setfill;
7
   usi ng std::setw;
8
9
10 #include "Time, h" // include definition of class Time
11
12 // constructor function to initialize private data;
13 // calls member function setTime to set variables:
14 // default values are 0 (see class definition)
15 Time::Time( int hour, int minute, int second )
16 {
      setTime( hour, minute, second );
17
18 } // end Time constructor
19
20 // set hour, minute and second values
21 void Time::setTime( int hour, int minute, int second )
22 {
23
      setHour( hour );
24
      setMinute( minute );
25
      setSecond( second );
26 } // end function setTime
```



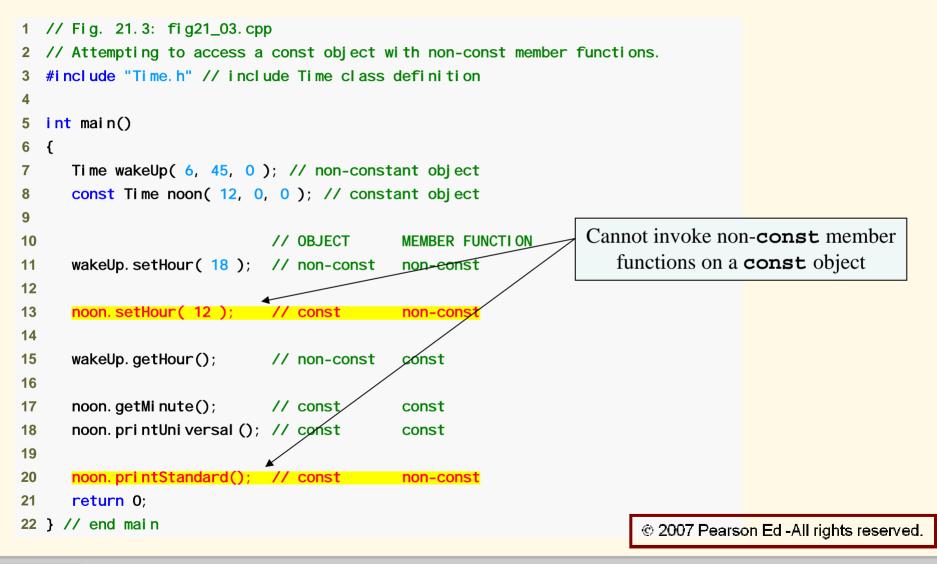
```
28 // set hour value
29 void Time::setHour( int h )
30 {
     hour = (h \ge 0 \& h < 24)? h : 0; // validate hour
31
32 } // end function setHour
33
34 // set minute value
35 void Time::setMinute( int m )
36 {
     minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute</pre>
37
38 } // end function setMinute
39
40 // set second value
41 void Time::setSecond(int s)
42 {
     second = (s \ge 0 \&\& s < 60) ? s : 0; // validate second
43
44 } // end function setSecond
                                                      const keyword in function definition,
45
                                                          as well as in function prototype
46 // return hour value
47 int Time::getHour() const // get functions should be const
48 {
      return hour;
                                                                      © 2007 Pearson Ed -All rights reserved.
49
50 } // end function getHour
```

<u>WPI</u>

27

```
51
52 // return minute value
53 int Time::getMinute() const
54 {
      return minute;
55
56 } // end function getMinute
57
58 // return second value
59 int Time::getSecond() const
60 {
61
      return second;
62 } // end function getSecond
63
64 // print Time in universal-time format (HH: MM: SS)
65 void Time:: printUniversal() const
66 {
      cout << setfill('0') << setw(2) << hour << ":"
67
         << setw( 2 ) << minute << ":" << setw( 2 ) << second;
68
69 } // end function printUniversal
70
71 // print Time in standard-time format (HH: MM: SS AM or PM)
72 void Time::printStandard() // note lack of const declaration
73 {
74
      cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
         << ":" << setfill( '0' ) << setw( 2 ) << minute
75
         << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );</pre>
76
                                                                              © 2007 Pearson Ed -All rights reserved.
77 } // end function printStandard
```









Borland C++ command-line compiler error messages:

Warning W8037 fig21_03.cpp 13: Non-const function Time::setHour(int)
 called for const object in function main()
Warning W8037 fig21_03.cpp 20: Non-const function Time::printStandard()
 called for const object in function main()

Microsoft Visual C++.NET compiler error messages:

C:\examples\ch21\Fig21_01_03\fig21_03.cpp(13) : error C2662: 'Time::setHour' : cannot convert 'this' pointer from 'const Time' to 'Time &' Conversion loses qualifiers C:\examples\ch21\Fig21_01_03\fig21_03.cpp(20) : error C2662: 'Time::printStandard' : cannot convert 'this' pointer from 'const Time' to 'Time &'

Conversion loses qualifiers

GNU C++ compiler error messages:

Fig21_03.cpp:13: error: passing `const Time' as `this' argument of `void Time::setHour(int)' discards qualifiers Fig21_03.cpp:20: error: passing `const Time' as `this' argument of `void Time::printStandard()' discards qualifiers

© 2007 Pearson Ed -All rights reserved.

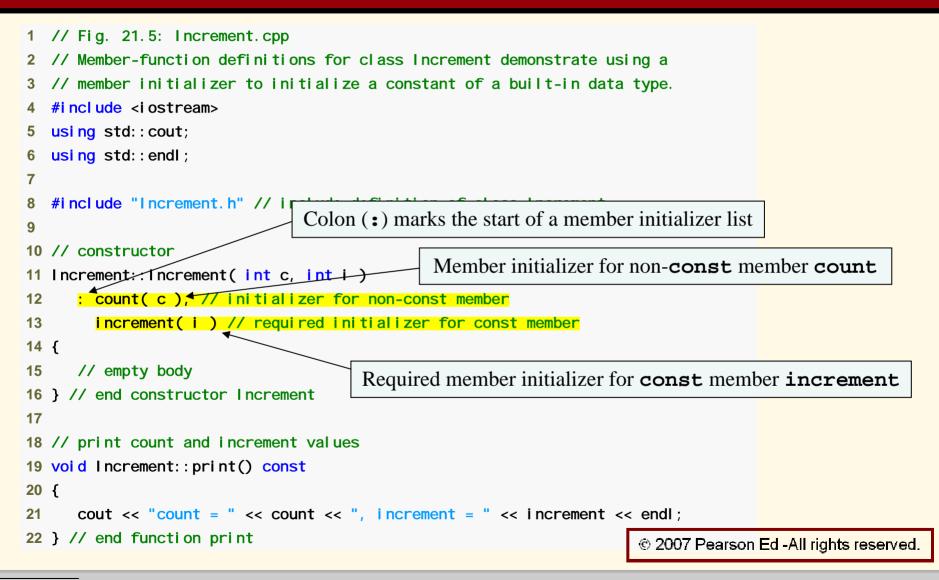


- Required for initializing
 - const data members
 - Data members that are references.
- . Can be used for any data member.
- Member initializer list
 - Appears between a constructor's parameter list and the left brace that begins the constructor's body.
 - Separated from the parameter list with a colon (:).
 - Each member initializer consists of the data member name followed by parentheses containing the member's initial value.
 - Multiple member initializers are separated by commas.
 - Executes before the body of the constructor executes.



```
1 // Fig. 21.4: Increment.h
  // Definition of class Increment.
2
  #ifndef INCREMENT H
3
  #define INCREMENT_H
4
5
6
  class Increment
  {
7
  public:
8
      Increment( int c = 0, int i = 1 ); // default constructor
9
10
11
      // function addincrement definition
     voi d addl ncrement()
12
13
      {
         count += increment;
14
      } // end function addIncrement
15
16
17
      void print() const; // prints count and increment
                                                               const data member that must be
18 private:
      int count:
                                                             initialized using a member initializer
19
      const int increment; // const data member
20
21 }; // end class Increment
22
                                                                    © 2007 Pearson Ed -All rights reserved.
23 #endif
```





WPI

```
1 // Fig. 21.6: fig21_06.cpp
2 // Program to test class Increment.
3 #include <iostream>
4 using std::cout;
5
  #include "Increment, h" // include definition of class Increment
6
7
8 int main()
9 {
      Increment value( 10, 5 );
10
11
12
      cout << "Before incrementing: ";</pre>
13
      value.print();
14
15
      for ( int | = 1; | <= 3; |++ )
16
      {
         value. addl ncrement();
17
         cout << "After increment " << j << ": ";</pre>
18
19
         value.print();
20
      } // end for
21
22
      return 0;
23 } // 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



Software Engineering Observation 21.3

 A const object cannot be modified by assignment, so it must be initialized. When a data member of a class is declared const, a member initializer must be used to provide the constructor with the initial value of the data member for an object of the class. The same is true for references.



Common Programming Error 21.5

 Not providing a member initializer for a const data member is a compilation error.



Software Engineering Observation 21.4

 Constant data members (const objects and const variables) and data members declared as references must be initialized with member initializer syntax; assignments for these types of data in the constructor body are not allowed



21.3 Composition: Objects as Members of Classes

- Composition

- Sometimes referred to as a *has-a* relationship.
- A class can have objects of other classes as members.
- Example
 - Al armCl ock object with a Time object as a member



Composition: Objects as Members of Classes

- . Initializing member objects
 - Member initializers pass arguments from the object's constructor to member-object constructors.
 - Member objects are constructed in the order in which they are declared in the class definition.
 - Not in the order they are listed in the constructor's member initializer list.
 - Before the enclosing class object (host object) is constructed.
 - If a member initializer is not provided
 - The member object's default constructor will be called implicitly.



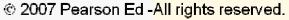
Software Engineering Observation 21.5

A common form of software reusability is composition, in which a class has objects of other classes as members.





```
1 // Fig. 21.10: Date.h
  // Date class definition; Member functions defined in Date.cpp
2
  #ifndef DATE H
3
  #define DATE H
4
5
  class Date
6
7
   {
  public:
8
9
      Date( int = 1, int = 1, int = 1900 ); // default constructor
      void print() const; // print date in month/day/year format
10
      ~Date(); // provided to confirm destruction order
11
12 private:
13
      int month; // 1-12 (January-December)
      int day; // 1-31 based on month
14
15
      int year; // any year
16
17
      // utility function to check if day is proper for month and year
18
      int checkDay( int ) const;
19 }; // end class Date
20
21 #endif
```





```
1 // Fig. 21.11: Date.cpp
  // Member-function definitions for class Date.
2
  #include <i ostream>
3
  usi ng std::cout;
4
  using std::endl;
5
6
   #include "Date.h" // include Date class definition
7
8
  // constructor confirms proper value for month; calls
9
10 // utility function checkDay to confirm proper value for day
11 Date::Date( int mn, int dy, int yr )
12 {
13
      if (mn > 0 \& km <= 12) // validate the month
         month = mn:
14
15
      el se
16
      {
         month = 1; // invalid month set to 1
17
         cout << "Invalid month (" << mn << ") set to 1. \n";</pre>
18
      } // end el se
19
20
      year = yr; // could validate yr
21
      day = checkDay( dy ); // validate the day
22
23
      // output Date object to show when its constructor is called
24
      cout << "Date object constructor for date ";</pre>
25
      print();
26
      cout << endl;
27
28 } // end Date constructor
```



```
29
30 // print Date object in form month/day/year
31 void Date::print() const
32 {
      cout << month << '/' << day << '/' << year;
33
34 } // end function print
35
36 // output Date object to show when its destructor is called
37 Date: : ~Date()
38 {
39
      cout << "Date object destructor for date ";
      print();
40
   cout << endl;
41
42 } // end ~Date destructor
```



```
43
44 // utility function to confirm proper day value based on
45 // month and year; handles leap years, too
46 int Date::checkDay( int testDay ) const
47 {
     static const int daysPerMonth[ 13 ] =
48
49
        { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
50
51
     // determine whether testDay is valid for specified month
52
     if (testDay > 0 && testDay <= daysPerMonth[month])
53
        return testDay;
54
55
     // February 29 check for leap year
56
     if (month == 2 && testDay == 29 && (year % 400 == 0 ||
         ( year % 4 == 0 && year % 100 != 0 ) ) )
57
58
        return testDay;
59
     cout << "Invalid day (" << testDay << ") set to 1. \n";
60
61
     return 1; // leave object in consistent state if bad value
62 } // end function checkDay
```



1	// Fig. 21.12: Employee.h		
2	// Employee class definition.		
3	3 // Member functions defined in Employee.cpp.		
4	#ifndef EMPLOYEE_H		
5	#define EMPLOYEE_H		
6			
7	/ #include "Date.h" // include Date class definition		
8			
9	class Employee	D (1	1 . 1
10		Parameters to be passe	
11	public:	nitializers to the construct	or for class Date
12	2 Employee(const char * const, const char * const,		
13	const Date &, const Date &);		
14	<pre>void print() const;</pre>		
15	<pre>~Employee(); // provided to confirm</pre>	de de la chieste ef elev	Determinenter
16	pri vate:	const objects of clas	ss Date as members
17	<pre>char firstName[25];</pre>		
18	B char lastName[25];		
19	9 const Date birthDate; 1/ composition: member object		
20	20 const Date hireDate; 1// composition: member object		
21	<pre>}; // end class Employee</pre>		
22		1	© 2007 Pearson Ed -All rights reserved.
23	#endif		



```
1 // Fig. 21.13: Employee.cpp
  // Member-function definitions for class Employee.
3
  #include <iostream>
   usi ng std::cout;
4
  using std::endl;
5
6
  #include <cstring> // strlen and strncpy prototypes
7
   using std::strlen;
8
  using std::strncpy;
9
10
11 #include "Employee.h" // Employee class definition
12 #include "Date.h" // Date class definition
13
14 // constructor uses member initializer list to pass initializer
15 // values to constructors of member objects birthDate and hireDate
16 // [Note: This invokes the so-called "default copy constructor" which the
17 // C++ compiler provides implicitly.]
18 Employee::Employee( const char * const first, const char * const last,
      const Date &dateOfBirth, const Date &dateOfHire )
19
      : birthDate( dateOfBirth ), \checkmark initialize birthDate
20
        hireDate( dateOfHire ) 4/ i ni ti al i ze hi reDate
21
                                                                Member initializers that pass arguments to
22 {
                                                                 Date's implicit default copy constructor
      // copy first into firstName and be sure that it fits
23
     int length = strlen( first );
24
     length = (length < 25 ? length : 24 );
25
26
      strncpy( firstName, first, length );
                                                                                   © 2007 Pearson Ed -All rights reserved.
     firstName[length] = '\0';
27
```



```
28
29
      // copy last into lastName and be sure that it fits
      length = strlen( last );
30
31
      length = (length < 25 ? length : 24);
32
      strncpy(lastName, last, length);
33
      lastName[ length ] = '\0';
34
35
      // output Employee object to show when constructor is called
36
      cout << "Employee object constructor: "</pre>
         << firstName << ' ' << lastName << endl;
37
38 } // end Employee constructor
39
40 // print Employee object
41 void Employee::print() const
42 {
      cout << lastName << ", " << firstName << " Hired: ";</pre>
43
     hi reDate. pri nt();
44
     cout << " Bi rthday: ";
45
     bi rthDate. pri nt();
46
      cout << endl;
47
48 } // end function print
49
50 // output Employee object to show when its destructor is called
51 Employee: : ~ Employee()
52 {
      cout << "Employee object destructor: "</pre>
53
         << lastName << ", " << firstName << endl;</pre>
54
55 } // end ~Employee destructor
```

© 2007 Pearson Ed -All rights reserved.



```
1 // Fig. 21.14: fig21_14.cpp
  // Demonstrating composition--an object with member objects.
2
   #include <iostream>
3
   usi ng std::cout;
4
   using std::endl;
5
6
   #include "Employee.h" // Employee class definition
7
8
  int main()
9
10 {
      Date birth( 7, 24, 1949 );
11
12
      Date hire( 3, 12, 1988 );
      Employee manager( "Bob", "Blue", birth, hire );
13
14
      cout << endl:
15
                                                             Passing objects to a host object constructor
      manager.print();
16
17
18
      cout << "\nTest Date constructor with invalid values: \n";
19
      Date lastDayOff( 14, 35, 1994 ); // invalid month and day
      cout << endl;
20
      return 0:
21
    // end main
22 }
                                                                           © 2007 Pearson Ed -All rights reserved.
```



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



Common Programming Error 21.6

- A compilation error occurs if a member object is not initialized with a member initializer and the member object's class does not provide a default constructor (i.e., the member object's class defines one or more constructors, but none is a default constructor).



21.4 fri end Functions and fri end Classes

• fri end function of a class

- Defined outside that class's scope.
- Not a member function of that class.
- has the right to access the non-public and public members of that class.
- Standalone functions or entire classes may be declared to be friends of a class.
- Can enhance performance.
- Often appropriate when a member function cannot be used for certain operations.



friend Functions and friend Classes

- . To declare a function as a friend of a class:
 - Provide the function prototype in the class definition preceded by keyword friend.
- To declare a class as a friend of another class:
 - Place a declaration of the form

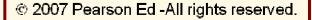
fri end cl ass Cl assTwo; in the definition of class Cl assOne

 All member functions of class Cl assTwo are fri ends of class Cl assOne.



fri end Functions and fri end Classes

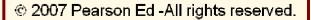
- · Friendship is granted, not taken.
 - For class B to be a friend of class A, class A must explicitly declare that class B is its friend.
- Friendship relation is neither symmetric nor transitive
 - If class A is a friend of class B, and class B is a friend of class C, you cannot infer that class B is a friend of class A, that class C is a friend of class B, or that class A is a friend of class C.





fri end Functions and fri end Classes

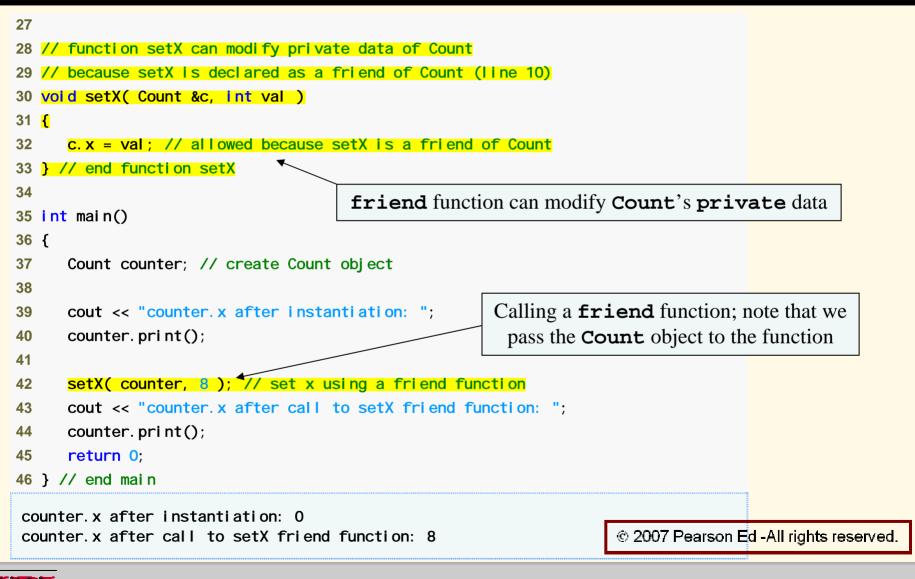
- It is possible to specify overloaded functions as fri ends of a class.
 - Each overloaded function intended to be a fri end must be explicitly declared as a fri end of the class.





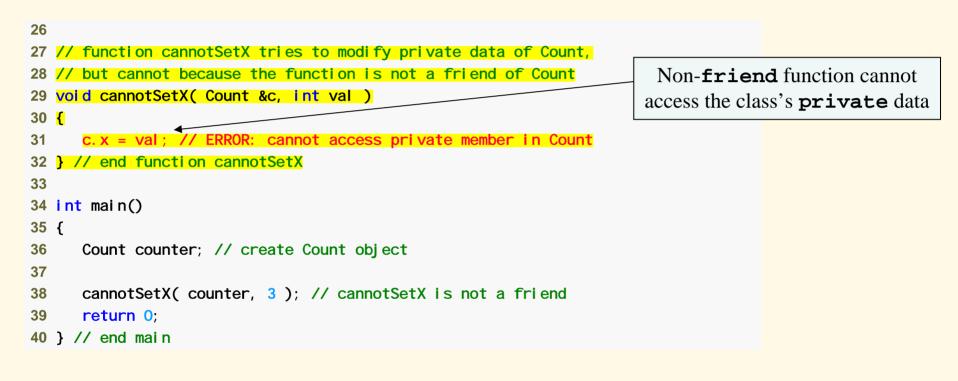
```
1 // Fig. 21.15: fig21_15.cpp
2 // Friends can access private members of a class.
  #include <iostream>
3
  usi ng std::cout;
4
  using std::endl;
5
6
                                          friend function declaration (can
  // Count class definition
7
                                             appear anywhere in the class)
  class Count
8
9
  {
      friend void setX( Count &, int ); // friend declaration
10
11 public:
12
      // constructor
13
      Count()
         : x( 0 ) // initialize x to 0
14
      {
15
         // empty body
16
      } // end constructor Count
17
18
     // output x
19
      void print() const
20
21
      {
22
         cout << x << endl;
      } // end function print
23
24 private:
      int x; // data member
25
                                                                                © 2007 Pearson Ed -All rights reserved.
26 }; // end class Count
```

WP



```
1 // Fig. 10.16: fig10_16.cpp
2 // Non-friend/non-member functions cannot access private data of a class.
3
  #i ncl ude <i ostream>
4 using std::cout;
  using std::endl;
5
6
  // Count class definition (note that there is no friendship declaration)
7
  class Count
8
  {
9
10 public:
     // constructor
11
     Count()
12
        : x(0) // initialize x to 0
13
     {
14
         // empty body
15
      } // end constructor Count
16
17
     // output x
18
     void print() const
19
      {
20
         cout << x << endl;
21
      } // end function print
22
23 private:
                                                                   © 2007 Pearson Ed -All rights reserved.
24
     int x; // data member
25 }; // end class Count
```







Borland C++ command-line compiler error message:

```
Error E2247 Fig21_16/fig21_16.cpp 31: 'Count::x' is not accessible in
function cannotSetX(Count &, int)
```

Microsoft Visual C++.NET compiler error messages:

C: \exampl es\ch21\Fig21_16\fig21_16. cpp(31) : error C2248: 'Count::x' : cannot access private member declared in class 'Count' C: \exampl es\ch21\Fig21_16\fig21_16. cpp(24) : see declaration of 'Count::x' C: \exampl es\ch21\Fig21_16\fig21_16. cpp(9) : see declaration of 'Count'

GNU C++ compiler error messages:

Fig21_16.cpp:24: error: 'int Count::x' is private Fig21_16.cpp:31: error: within this context



21.5 Using the this Pointer

- 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).
 - An object's this pointer is not part of the object itself.
 - The this pointer is passed (by the compiler) as an implicit argument to each of the object's non-static member functions.



21.5 Using the this Pointer

- Objects use the this pointer implicitly or explicitly.
 - Used implicitly when accessing members directly.
 - Used explicitly when using keyword this.
 - Type of the this pointer depends on the type of the object and whether the executing member function is declared **const**.

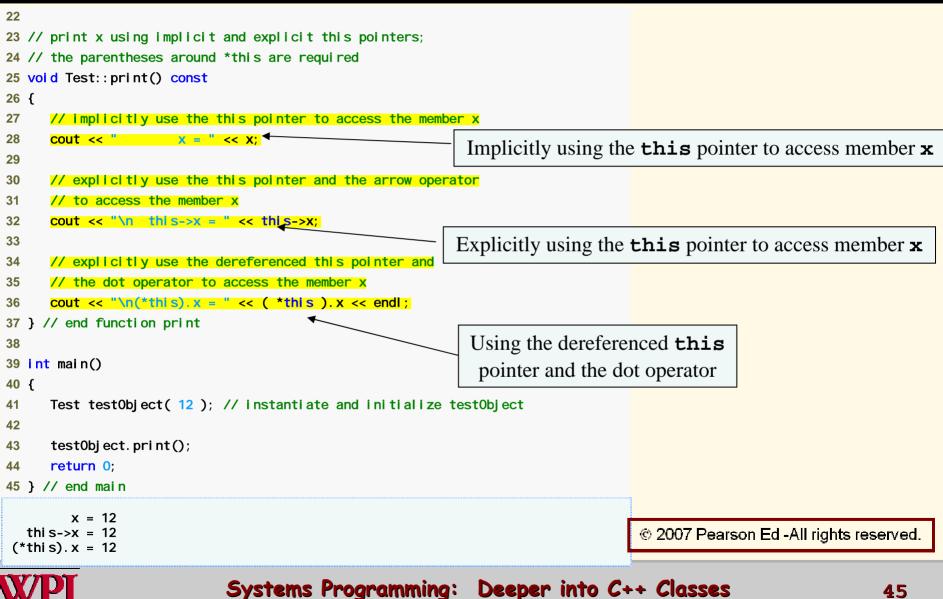


this Example

```
1 // Fig. 21.17: fig21_17.cpp
2 // Using the this pointer to refer to object members.
  #i ncl ude <i ostream>
3
  usi ng std::cout;
4
5 using std::endl;
6
  class Test
7
8
  {
9
  public:
     Test( int = 0 ); // default constructor
10
     void print() const;
11
12 pri vate:
13
   int x;
14 }; // end class Test
15
16 // constructor
17 Test::Test( int value )
     : x(value) // initialize x to value
18
19 {
     // empty body
20
21 } // end constructor Test
```



this Example



Common Programming Error 21.7

. Attempting to use the member selection operator (.) with a pointer to an object is a compilation error the dot member selection operator may be used only with an Ivalue such as an object's name, a reference to an object or a dereferenced pointer to an object.



Using the this Pointer

- Cascaded member-function calls
 - Multiple functions are invoked in the same statement.
 - Enabled by member functions returning the dereferenced this pointer
 - Example
 - t.setMinute(30).setSecond(22);
 - Calls t. setMi nute(30);
 - Then calls t. setSecond(22);



```
1 // Fig. 21.18: Time.h
  // Cascading member function calls.
2
3
  // Time class definition.
4
  // Member functions defined in Time.cpp.
  #ifndef TIME H
6
  #define TIME H
7
8
  class Time
Q
10 {
11 public:
12
      Time( int = 0, int = 0, int = 0 ); // default constructor
13
     // set functions (the Time & return types enable cascading)
14
15
      Time &setTime( int, int, int ); // set hour, minute, second
      Time &setHour( int ); // set hour
16
17
      Time &setMinute( int ); // set minute
18
      Time &setSecond( int ); // set second
                           set functions return Time & to enable cascading
```



```
19
20
      // get functions (normally declared const)
21
      int getHour() const; // return hour
      int getMinute() const; // return minute
22
      int getSecond() const; // return second
23
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)
      int minute: // 0 - 59
30
31
      int second; // 0 - 59
32 }: // end class Time
33
34 #endif
```



```
1 // Fig. 21.19: Time.cpp
2 // Member-function definitions for Time class.
   #include <iostream>
3
   usi ng std::cout;
4
5
  #i ncl ude <i omani p>
6
   using std::setfill;
7
   usi ng std::setw;
8
9
10 #include "Time, h" // Time class definition
11
12 // constructor function to initialize private data;
13 // calls member function setTime to set variables;
14 // default values are 0 (see class definition)
15 Time::Time( int hr, int min, int sec )
16 {
      setTime( hr, min, sec );
17
18 } // end Time constructor
19
20 // set values of hour, minute, and second
21 Time & Time::setTime(int h, int m, int s) // note Time & return
22 {
      setHour( h );
23
                                               Returning dereferenced this pointer enables cascading
      setMinute( m );
24
25
      setSecond( s ); 
      return *this; // enables cascading
                                                                          © 2007 Pearson Ed -All rights reserved.
26
27 } // end function setTime
```

Systems Programming: Deeper into C++ Classes

50



```
29 // set hour value
30 Time & Time::setHour(int h) // note Time & return
31 {
     hour = (h \ge 0 \& h < 24)? h : 0; // validate hour
32
33
     return *this: // enables cascading
34 } // end function setHour
35
36 // set minute value
37 Time & Time::setMinute(int m) // note Time & return
38 {
     minute = (m \ge 0 \& \& m < 60) ? m : 0; // validate minute
39
     return *this; // enables cascading
40
41 } // end function setMinute
42
43 // set second value
44 Time & Time::setSecond(ints) // note Time & return
45 {
46
     second = (s \ge 0 \&\& s < 60) ? s : 0; // validate second
     return *this: // enables cascading
47
48 } // end function setSecond
49
50 // get hour value
51 int Time::getHour() const
52 {
53
      return hour;
54 } // end function getHour
```

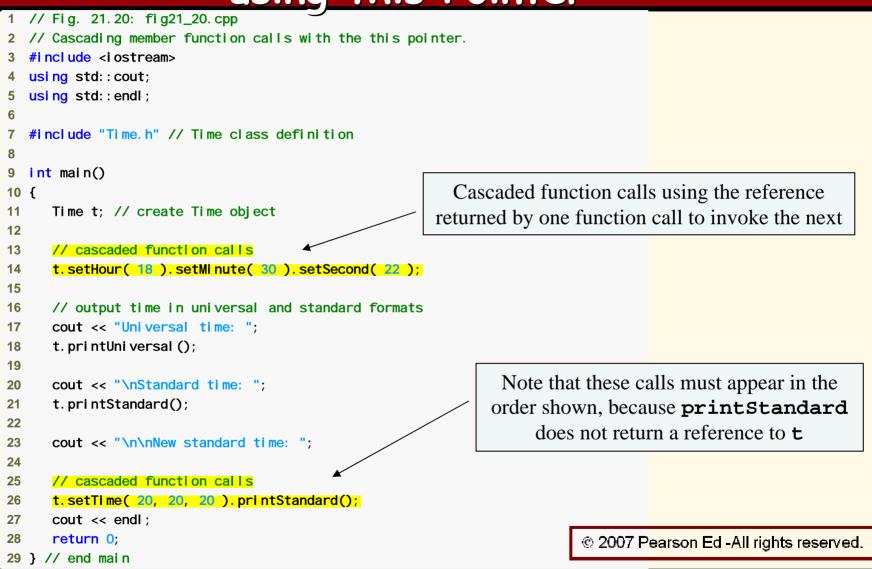
© 2007 Pearson Ed -All rights reserved.



28

```
55
56 // get minute value
57 int Time::getMinute() const
58 {
      return minute:
59
60 } // end function getMinute
61
62 // get second value
63 int Time::getSecond() const
64 {
65
      return second;
66 } // end function getSecond
67
68 // print Time in universal-time format (HH: MM: SS)
69 void Time:: printUniversal() const
70 {
71
      cout << setfill('0') << setw(2) << hour << ":"
         << setw( 2 ) << minute << ":" << setw( 2 ) << second;
72
73 } // end function printUniversal
74
75 // print Time in standard-time format (HH: MM: SS AM or PM)
76 void Time::printStandard() const
77 {
78
      cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
         << ":" << setfill( '0' ) << setw( 2 ) << minute</pre>
79
         << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );</pre>
                                                                                 © 2007 Pearson Ed -All rights reserved.
80
81 } // end function printStandard
```







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

New standard time: 8:20:20 PM



21.6 Dynamic Memory Management:Operators new and delete

- Dynamic memory management
 - Enables programmers to allocate and deallocate memory for any built-in or user-defined type.
 - Performed by operators new and del ete.
 - For example, dynamically allocating memory for an array instead of using a fixed-size array.



- · Operator new
 - Allocates (i.e., reserves) storage of the proper size for an object at execution time
 - Calls a constructor to initialize the object.
 - Returns a pointer of the type specified to the right of new.
 - Can be used to dynamically allocate any fundamental type (such as int or double) or any class type.
- . The Free store (referred to as the heap)
 - Region of memory assigned to each program for storing objects created at execution time.

```
Example:
```

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



. Operator del ete

- Destroys a dynamically allocated object.
- Calls the destructor for the object.
- Deallocates (i.e., releases) memory from the free store.
- The memory can then be reused by the system to allocate other objects.

Example:

delete timePtr;



- . Initializing an object allocated by new
 - Initializer for a newly created fundamental-type variable.
 - Example

- doubl e *ptr = new doubl e(3.14159);

- Specify a comma-separated list of arguments to the constructor of an object.
 - Example

- Time *timePtr = new Time(12, 45, 0);



- new operator can be used to allocate arrays dynamically.
 - Dynamically allocate a 10-element integer array:
 - int *gradesArray = new int[10];
 - Size of a dynamically allocated array
 - Specified using any integral expression that can be evaluated at execution time.
 mulePtr *Mule = new Mules[mules_in];



- Delete a dynamically allocated array: del ete [] gradesArray;
 - This deallocates the array to which gradesArray points.
 - If the pointer points to an array of objects,
 - It first calls the destructor for every object in the array.
 - Then it deallocates the memory.
 - If the statement did not include the square brackets ([]) and gradesArray pointed to an array of objects
 - Only the first object in the array would have a destructor call.



^{© 2007} Pearson Ed -All rights reserved.

- static data member
 - Only one copy of a variable shared by all objects of a class.
 - The member is "Class-wide" information.
 - A property of the class shared by all instances, not a property of a specific object of the class.
 - Declaration begins with keyword static



^{© 2007} Pearson Ed -All rights reserved.

- Example

- Video game with Marti ans and other space creatures
 - Each Marti an needs to know the marti anCount.
 - marti anCount should be stati c class-wide data.
 - Every Marti an can access marti anCount as if it were a data member of that Marti an
 - Only one copy of marti anCount exists.
- May seem like global variables but static has class scope.
- Can be declared public, private or protected. © 2007 Pearson Ed - All rights reserved.



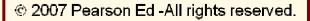
- Fundamental-type static data members
 - Initialized by default to 0.
 - If you want a different initial value, a static data member can be initialized once (and only once).
- const static data member of int or enum type
 - Can be initialized in its declaration in the class definition.
- · All other static data members
 - Must be defined at file scope (i.e., outside the body of the class definition)
 - Can be initialized only in those definitions.
- static data members of class types (i.e., static member objects) that have default constructors
 - Need not be initialized because their default constructors will be called.



- · Exists even when no objects of the class exist.
 - To access a public static class member when no objects of the class exist.
 - Prefix the class name and the binary scope resolution operator (::) to the name of the data member.
 - Example
 - » Marti an: : marti anCount
 - Also accessible through any object of that class
 - Use the object's name, the dot operator and the name of the member.
 - Example
 - » myMarti an. marti anCount



- static member function
 - Is a service of the *class*, not of a specific object of the class.
- static is applied to an item at file scope.
 - That item becomes known only in that file.
 - The static members of the class need to be available from any client code that accesses the file.
 - So we cannot declare them static in the . cpp file we declare them static only in the . h file.





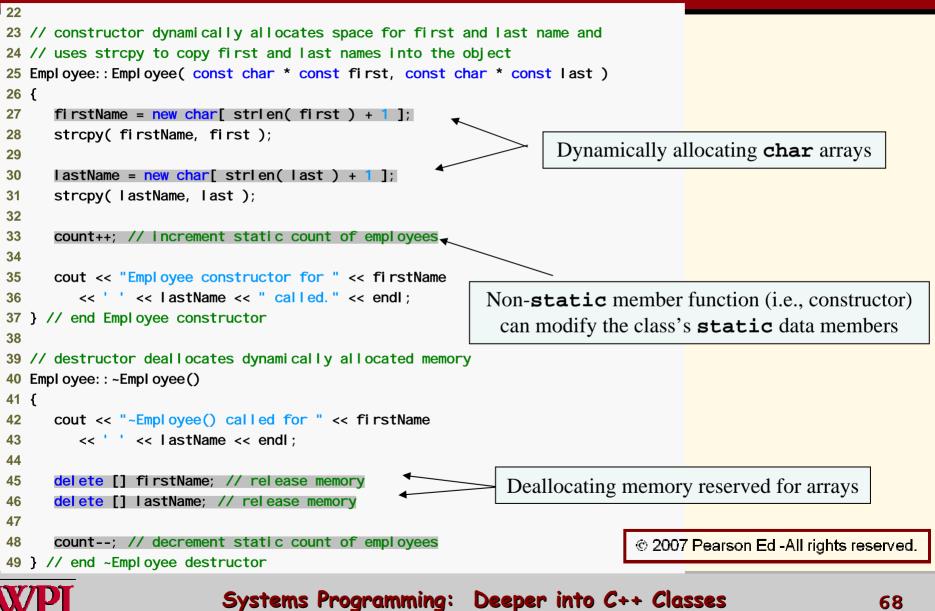
```
1 // Fig. 21.21: Employee.h
  // Employee class definition.
   #ifndef EMPLOYEE H
  #define EMPLOYEE H
5
  class Employee
6
7
   {
  public:
8
      Employee( const char * const, const char * const ); // constructor
9
      ~Employee(); // destructor
10
     const char *getFirstName() const; // return first name
11
      const char *getLastName() const; // return last name
12
13
      // static member function
14
15
      static int getCount(); // return number of objects instantiated
16 pri vate:
                                                     Function prototype for static member function
      char *firstName;
17
      char *lastName;
18
19
20
      // static data
                                                           static data member keeps track of number
      static int count; // number of objects instantiated
21
                                                             of Employee objects that currently exist
22 }; // end class Employee
                                                                       © 2007 Pearson Ed -All rights reserved.
23
24 #endif
                        Systems Programming: Deeper into C++ Classes
                                                                                                    66
```

1	// Fig. 21.22: Employee.cpp				
2	// Member-function definitions for class Employee.				
3	<pre>#include <iostream></iostream></pre>				
4	using std::cout;				
5	using std::endl;				
6					
7	<pre>#include <cstring> // strlen and strcpy prototypes</cstring></pre>				
8	usi ng std: : strl en;				
9	using std::strcpy;				
10					
11	<pre>#include "Employee.h" // Employee class definition</pre>				
12					
13	// define and initialize static data member at file scope		1 /	1 1 1 1 1	
14	<pre>int Employee::count = 0;</pre>			mber is defined and	
15		initial	ized at file sc	ope in the .cpp file	
16	// define static member function that returns number of				
17	7 // Employee objects instantiated (declared static in Employee.h)				
18	int Employee::getCount()	a+ a		function con access	
19	(static member function can access			
20	return count;	-		, because the function	
21	<pre>} // end static function getCount</pre>	might be called when no objects exist			
			© 2007 Pears	on Ed -All rights reserved.	

Systems Programming: Deeper into C++ Classes

67



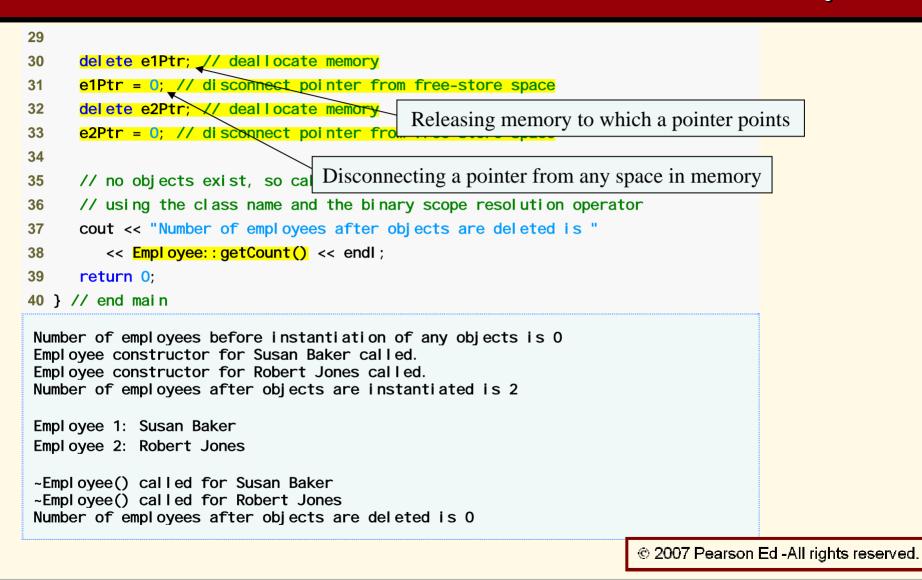


```
50
51 // return first name of employee
52 const char *Employee::getFirstName() const
53 {
     // const before return type prevents client from modifying
54
     // private data; client should copy returned string before
55
56
     // destructor deletes storage to prevent undefined pointer
     return firstName:
57
58 } // end function getFirstName
59
60 // return last name of employee
61 const char *Employee::getLastName() const
62 {
     // const before return type prevents client from modifying
63
     // private data; client should copy returned string before
64
     // destructor deletes storage to prevent undefined pointer
65
66
     return lastName:
67 } // end function getLastName
```



1 // Fig. 21.23: fig21_23.cpp					
2 // Driver to test class Employee.					
3 #include <iostream></iostream>					
4 using std::cout;					
5 using std::endl;					
6					
7 #include "Employee.h" // Employee class definition					
8					
9 int main()					
10 {					
11 // use class name and binary scope resolution operator to					
12 // access sta tic number function getCount					
13 cout << "Number of employees before instantiation of any objects is "					
14 << Employee::getCount() << endl; // use class name					
15	Calling static member function using class				
16 // use new to dynamically create two new Employees	name and binary scope resolution operator				
17 // operator new also calls the object's constructor					
18 Employee *e1Ptr = new Employee("Susan", "Baker");					
19 Employee *e2Ptr = new Employee("Robert", "Jones");	Dynamically creating Employees with new				
20					
21 // call getCount on first Employee object					
22 cout << "Number of employees after objects are instantiate	dis_"				
23 << e1Ptr->getCount();	Calling a static member function				
24					
25 cout << "\n\nEmployee 1: "	through a pointer to an object of the class				
<pre>26 << e1Ptr->getFirstName() << " " << e1Ptr->getLastName()</pre>					
27 << "\nEmployee 2: "	© 200 <mark>7 Pearson Ed -All rights reserved.</mark>				
<pre>28 << e2Ptr->getFirstName() << " " << e2Ptr->getLastName()</pre>	<< "\n\n";				







- Declare a member function static
 - If it does not access non-static data members or non-static member functions of the class.
- A static member function does not have a this pointer.
- static data members and static member functions exist independently of any objects of a class.
- When a static member function is called, there might not be any objects of its class in memory.



^{© 2007} Pearson Ed -All rights reserved.

Abstract data types (ADTs)

- Essentially ways of representing real-world notions to some satisfactory level of precision within a computer system.
- Types like int, double, char and others are all ADTs.
 - e.g., int is an abstract representation of an integer.
- Captures two notions:
 - Data representation
 - Operations that can be performed on the data.
- C++ classes implement ADTs and their services.



Array Abstract Data Type

- Many array operations not built into C++
 - e.g., subscript range checking
- Programmers can develop an array ADT as a class that is preferable to "raw" arrays
 - Can provide many helpful new capabilities
- C++ Standard Library class template vector.



Summary

- const objects and const member functions
- Member Composition Example
- Friend function Example
- this pointer Example
- Dynamic memory management
 - new and del ete operators
- static class members
- Abstract Data Types

