

```

1 // Fig. 6.9: salesp.h
2 // SalesPerson class definition.
3 // Member functions defined in salesp.cpp.
4 #ifndef SALESP_H
5 #define SALESP_H
6
7 class SalesPerson {
8
9 public:
10     SalesPerson();           // construct
11     void getSalesFromUser(); // input sales from keyboard
12     void setSales( int, double ); // set sales
13     void printAnnualSales(); // summarize
14
15 private:
16     double totalAnnualSales(); // utility function
17     double sales[ 12 ];        // 12 monthly sales figures
18
19 }; // end class SalesPerson
20
21 #endif

```

Set access function performs validity checks.

private utility function.

Figure 1: SalesPerson class definition

1 Access Functions and Utility Functions

Not all member functions need be made **public** to serve as part of the interface of the class.

- Access functions
- **public**
 - Read/display data
 - Predicate functions
 - Check conditions
 - Utility functions (helper functions)
- **private**
 - Support operation of **public** member functions
 - Not intended for direct client use

The program of Figs. 1-4 demonstrates the notion of a *utility function* (also called helper function).

```

1 // Fig. 6.10: salesp.cpp
2 // Member functions for class SalesPerson.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8 using std::fixed;
9
10 #include <iomanip>
11
12 using std::setprecision;
13
14 // include SalesPerson class definition from salesp.h
15 #include "salesp.h"
16
17 // initialize elements of array sales to 0.0
18 SalesPerson::SalesPerson()
19 {
20     for ( int i = 0; i < 12; i++ )
21         sales[ i ] = 0.0;
22 }
23 // end SalesPerson constructor
24

```



[Outline](#)

41

salesp.cpp (1 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

25 // get 12 sales figures from the user at the keyboard
26 void SalesPerson::getSalesFromUser()
27 {
28     double salesFigure;
29
30     for ( int i = 1; i <= 12; i++ ) {
31         cout << "Enter sales amount for month " << i << ": ";
32         cin >> salesFigure;
33         setSales( i, salesFigure );
34     }
35 // end for
36
37 // end function getSalesFromUser
38
39 // set one of the 12 monthly sales figures: function subtracts
40 // one from month value for proper subscript
41 void SalesPerson::setSales( int month, double amount )
42 {
43     // test for valid month and amount values
44     if ( month >= 1 && month <= 12 && amount > 0 )
45         sales[ month - 1 ] = amount; // adjust for subscripts 0-11
46
47     else // invalid month or amount value
48         cout << "Invalid month or sales figure" << endl;
49 }
50 // end function setSales
51

```

Set access function performs validity checks.



[Outline](#)

42

salesp.cpp (2 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 2: **SalesPerson** class member-function definitions (part 1 of 2)

```
49
50 } // end function setSales
51
52 // print total annual sales (with help of utility function)
53 void SalesPerson::printAnnualSales()
54 {
55     cout << setprecision( 2 ) << fixed
56         << "\nThe total annual sales are: $"
57         << totalAnnualSales() << endl; // call utility function
58
59 } // end function printAnnualSales
60
61 // private utility function to total annual sales
62 double SalesPerson::totalAnnualSales()
63 {
64     double total = 0.0;           // initialize total
65
66     for ( int i = 0; i < 12; i++ ) // summarize sales results
67         total += sales[ i ];
68
69     return total;
70
71 } // end function totalAnnualSales
```

Outline 43
salesp.cpp (3 of 3)

private utility function to help function printAnnualSales; encapsulates logic of manipulating sales array.

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 3: SalesPerson class member-function definitions (part 2 of 2)

```
1 // Fig. 6.11: fig06_11.cpp
2 // Demonstrating a utility function.
3 // Compile this program with salesp.cpp
4
5 // include SalesPerson class definition from salesp.h
6 #include "salesp.h"
7
8 int main()
9 {
10     SalesPerson s; // create SalesPerson object
11
12     s.getSalesFromUser(); // note simple sequential code; no
13     s.printAnnualSales(); // control structures in main
14
15     return 0;
16
17 }
```

Simple sequence of member function calls; logic encapsulated in member functions.

44
Outline
fig06_11.cpp
(1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

```
Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

The total annual sales are: $60120.59
```

45
Outline
fig06_11.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 4: Utility function demonstration

2 Initializing Class Objects: Constructors

- Constructors
 - Initialize data members; Or can set later
 - Same name as class
 - No return type
- Initializers
 - Passed as arguments to constructor
 - In parentheses to right of class name before semicolon

```
Class-type ObjectName( value1,value2,...};
```

The programmer provides the constructor, which is then invoked each time an object of that class is created (instantiated).

3 Using Default Arguments with Constructors

- Constructors
 - Can specify default arguments
 - Default constructors
 - Defaults all arguments
 - OR
 - Explicitly requires no arguments
 - Can be invoked with no arguments
 - Only one per class

The program of Figs. 5-9 enhances class **Time** to demonstrate how arguments are implicitly passed to a constructor.

```
1 // Fig. 6.12: time2.h
2 // Declaration of class Time.
3 // Member functions defined in time2.cpp.
4
5 // prevent multiple inclusions of header file
6 #ifndef TIME2_H
7 #define TIME2_H
8
9 // Time abstract data type definition
10 class Time {
11
12 public:
13     Time( int = 0, int = 0, int = 0); // default constructor
14     void setTime( int, int, int ); // set hour, minute, second
15     void printUniversal(); // print universal-time format
16     void printStandard(); // print standard-time format
17
18 private:
19     int hour; // 0 - 23 (24-hour clock format)
20     int minute; // 0 - 59
21     int second; // 0 - 59
22
23 }; // end class Time
24
25 #endif
```

Default constructor specifying all arguments.



Figure 5: **Time** class containing a constructor with default arguments.

```
1 // Fig. 6.13: time2.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 time2.h
13 #include "time2.h"
14
15 // Time constructor initializes each data member to zero;
16 // ensures all Time objects start in a consistent state
17 Time::Time( int hr, int min, int sec )
18 {
19     setTime( hr, min, sec ); // validate and set time
20 }
21 // end Time constructor
22
```

Constructor calls `setTime` to validate passed (or default) values.



Outline

49

time2.cpp (1 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.

```
23 // set new Time value using universal time, perform validity
24 // checks on the data values and set invalid values to zero
25 void Time::setTime( int h, int m, int s )
26 {
27     hour = ( h >= 0 && h < 24 ) ? h : 0;
28     minute = ( m >= 0 && m < 60 ) ? m : 0;
29     second = ( s >= 0 && s < 60 ) ? s : 0;
30 }
31 // end function setTime
32
33 // print Time in universal format
34 void Time::printUniversal()
35 {
36     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
37         << setw( 2 ) << minute << ":"
38         << setw( 2 ) << second;
39 }
40 // end function printUniversal
41
```



Outline

50

time2.cpp (2 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 6: **Time** class member-function definitions including a constructor that takes arguments. (part 1 of 2)

```
42 // print Time in standard format
43 void Time::printStandard()
44 {
45     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
46         << ":" << setfill( '0' ) << setw( 2 ) << minute
47         << ":" << setw( 2 ) << second
48         << ( hour < 12 ? " AM" : " PM" );
49
50 } // end function printStandard
```



Outline

time2.cpp (3 of 3)

Figure 7: **Time** class member-function definitions including a constructor that takes arguments. (part 2 of 2)


```

1 // Fig. 6.14: fig06_14.cpp
2 // Demonstrating a default constructor for class Time.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 // include definition of class Time from time2.h
9 #include "time2.h"
10
11 int main()
12 {
13     Time t1;           // all arguments defaulted
14     Time t2( 2 );     // minute and second defaulted
15     Time t3( 21, 34 ); // second defaulted
16     Time t4( 12, 25, 42 ); // all values specified
17     Time t5( 27, 74, 95 ); // all bad values specified
18
19     cout << "Constructed with:\n\n"
20          << "all default arguments:\n ";
21     t1.printUniversal(); // 00:00:00
22     cout << "\n ";
23     t1.printStandard(); // 12:00:00 AM
24

```

Initialize Time objects using default arguments.

Initialize Time object with invalid values; validity checking will set values to 0.

```

25     cout << "\n\nhour specified, default minute and second:\n ";
26     t2.printUniversal(); // 02:00:00
27     cout << "\n ";
28     t2.printStandard(); // 2:00:00 AM
29
30     cout << "\n\nhour and minute specified, default second:\n ";
31     t3.printUniversal(); // 21:34:00
32     cout << "\n ";
33     t3.printStandard(); // 9:34:00 PM
34
35     cout << "\n\nhour, minute, and second specified:\n ";
36     t4.printUniversal(); // 12:25:42
37     cout << "\n ";
38     t4.printStandard(); // 12:25:42 PM
39
40     cout << "\n\nall invalid values specified:\n ";
41     t5.printUniversal(); // 00:00:00
42     cout << "\n ";
43     t5.printStandard(); // 12:00:00 AM
44     cout << endl;
45
46     return 0;
47
48 } // end main

```

t5 constructed with invalid arguments; values set to 0.

Figure 8: Constructor with default arguments. (part 1 of 2)

```
Constructed with:

all default arguments:
00:00:00
12:00:00 AM

hour specified; default minute and second:
02:00:00
2:00:00 AM

hour and minute specified; default second:
21:34:00
9:34:00 PM

hour, minute, and second specified:
12:25:42
12:25:42 PM

all invalid values specified:
00:00:00
12:00:00 AM
```



[Outline](#)

54

fig06_14.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 9: Constructor with default arguments. (part 2 of 2)

4 Destructors

- Special member function
- Same name as class; Preceded with tilde (~)
- No arguments
- No return value
- Cannot be overloaded
- Performs "termination housekeeping"
 - Before system reclaims object's memory; Reuse memory for new objects
- No explicit destructor; Compiler creates "empty destructor"

5 When Constructors and Destructors Are Called

- Constructors and destructors; Called implicitly by compiler
- Order of function calls
 - Depends on order of execution; When execution enters and exits scope of objects
 - Generally, destructor calls reverse order of constructor calls
- Order of constructor, destructor function calls
 - Global scope objects
 - * Constructors; Before any other function (including **main**)
 - * Destructors
 - When **main** terminates (or **exit** function called)
 - Not called if program terminates with **abort**
 - Automatic local objects
 - * Constructors
 - When objects defined; Each time execution enters scope
 - * Destructors
 - When objects leave scope; Execution exits block in which object defined
 - Not called if program ends with **exit** or **abort**
 - **static** local objects
 - * Constructors
 - Exactly once
 - When execution reaches point where object defined
 - * Destructors
 - When **main** terminates or **exit** function called
 - Not called if program ends with **abort**

The program of Figs. 10-13 demonstrates the order in which constructors and destructors are called for objects of class `CreateAndDestroy` of various storage classes in several scopes.

```
1 // Fig. 6.15: create.h
2 // Definition of class CreateAndDestroy.
3 // Member functions defined in create.cpp.
4 #ifndef CREATE_H
5 #define CREATE_H
6
7 class CreateAndDestroy {
8
9 public:
10     CreateAndDestroy( int, char * ); // constructor
11     ~CreateAndDestroy();
12
13 private:
14     int objectID;
15     char *message;
16
17 }; // end class CreateAndDestroy
18
19 #endif
```

59

Outline

create.h (1 of 1)

Constructor and destructor member functions.

private members to show order of constructor, destructor function calls.

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 10: `CreateAndDestroy` class definition.

```
1 // Fig. 6.16: create.cpp
2 // Member-function definitions for class CreateAndDestroy
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 // include CreateAndDestroy class definition from create.h
9 #include "create.h"
10
11 // constructor
12 CreateAndDestroy::CreateAndDestroy(
13     int objectNumber, char *messagePtr )
14 {
15     objectID = objectNumber;
16     message = messagePtr;
17
18     cout << "Object " << objectID << " constructor runs "
19         << message << endl;
20
21 } // end CreateAndDestroy constructor
22
```

Output message to demonstrate timing of constructor function calls.

```
23 // destructor
24 CreateAndDestroy::~CreateAndDestroy()
25 {
26     // the following line is for pedagogy
27     cout << ( objectID == 1 ? "Object " : "Object " ) << objectID << " destructor runs "
28         << message << endl;
29
30 } // end ~CreateAndDestroy destructor
31
32
```

Output message to demonstrate timing of destructor function calls.

Figure 11: **CreateAndDestroy** class member-function definitions.

```

1 // Fig. 6.17: fig06_17.cpp
2 // Demonstrating the order in which constructors and
3 // destructors are called.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 // include CreateAndDestroy class definition from create.h
10 #include "create.h"
11
12 void create( void ); // prototype
13
14 // global object
15 CreateAndDestroy first( 1, "(global before main)" );
16
17 int main()
18 {
19     cout << "\nMAIN FUNCTION: EXECUTION" << endl;
20
21     CreateAndDestroy second( 2, "(local automatic in main)" );
22
23     static CreateAndDestroy third(
24         3, "(local static in main)" );
25

```

Create variable with global scope.

Create local automatic object.

Create static local object.



Outline

62

fig06_17.cpp
(1 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

26     create(); // call function to create objects
27
28     cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
29
30     CreateAndDestroy fourth( 4, "(local automatic objects)" );
31
32     cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
33
34     return 0;
35
36 } // end main
37
38 // function to create objects
39 void create( void )
40 {
41     cout << "\nCREATE FUNCTION" << endl;
42
43     CreateAndDestroy fifth( 5, "(local automatic in create)" );
44
45     static CreateAndDestroy sixth(
46         6, "(local static in create)" );
47
48     CreateAndDestroy seventh(
49         7, "(local automatic in create)" );
50

```

Create local automatic objects.

Create local automatic object.

Create local automatic object in function.

Create static local object in function.

Create local automatic object in function.



Outline

63

fig06_17.cpp
(2 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 12: Order in which constructors and destructors are called. (part 1 of 2)

```

51     cout << "\nCREATE FUNCTION: EXECUTION ENDS\n" << endl;
52
53 } // end function create

```



Outline

64

fig06_17.cpp
(3 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

Object 1  constructor runs  (global before main)
MAIN FUNCTION: EXECUTION BEGINS
Object 2  constructor runs  (local automatic in main)
Object 3  constructor runs  (local static in main)
CREATE FUNCTION: EXECUTION BEGINS
Object 5  constructor runs  (local automatic in create)
Object 6  constructor runs  (local static in create)
Object 7  constructor runs  (local automatic in create)
CREATE FUNCTION: EXECUTION ENDS
Object 7  destructor runs  (local automatic in create)
Object 5  destructor runs  (local automatic in create)
MAIN FUNCTION: EXECUTION RESUMES
Object 4  constructor runs  (local automatic in main)
MAIN FUNCTION: EXECUTION ENDS
Object 4  destructor runs  (local automatic in main)
Object 2  destructor runs  (local automatic in main)
Object 6  destructor runs  (local static in create)
Object 3  destructor runs  (local static in main)
Object 1  destructor runs  (global before main)

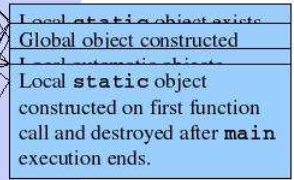
```



Outline

65

fig06_17.cpp
output (1 of 1)



© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 13: Order in which constructors and destructors are called. (part 2 of 2)

6 Using *Set* and *Get* Functions

A class's **private** data members can be accessed only by member functions (and friends) of the class. Classes often provide **public** member functions to allow clients of the class to *set* (i.e., write) or *get* (i.e., read) the values of **private** data members. These functions need not be called *set* and *get* specifically, but they often are.

- Set functions
 - Perform validity checks before modifying **private** data
 - Notify if invalid values
 - Indicate with return values
- Get functions
 - "Query" functions
 - Control format of data returned

The program of Figs. 14-18 enhances class **Time** to include *set* and *get* functions for the **private** data members **hour**, **minute**, and **second**.


```
1 // Fig. 6.18: time3.h
2 // Declaration of class Time.
3 // Member functions defined in time3.cpp
4
5 // prevent multiple inclusions of header file
6 #ifndef TIME3_H
7 #define TIME3_H
8
9 class Time {
10
11 public:
12     Time( int = 0, int = 0, int = 0 ); // default constructor
13
14     // set functions
15     void setTime( int, int, int ); // set hour, minute, second
16     void setHour( int ); // set hour
17     void setMinute( int ); // set minute
18     void setSecond( int ); // set second
19
20     // get functions
21     int getHour(); // return hour
22     int getMinute(); // return minute
23     int getSecond(); // return second
24
```

Set functions.

Get functions.

```
25     void printUniversal(); // output universal-time format
26     void printStandard(); // output standard-time format
27
28 private:
29     int hour; // 0 - 23 (24-hour clock format)
30     int minute; // 0 - 59
31     int second; // 0 - 59
32
33 }; // end clas Time
34
35 #endif
```

Figure 14: **Time** class definition with *set* and *get* functions.

```

1 // Fig. 6.19: time3.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 definition of class Time from time3.h
13 #include "time3.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 hr, int min, int sec )
19 {
20     setTime( hr, min, sec );
21 }
22 // end Time constructor
23

```



[Outline](#)

69

time3.cpp (1 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

24 // set hour, minute and second values
25 void Time::setTime( int h, int m, int s )
26 {
27     setHour( h );
28     setMinute( m );
29     setSecond( s );
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](#)

70

time3.cpp (2 of 4)

Call set functions to perform validity checking.

Set functions perform validity checks before modifying data.

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 15: **Time** class member-function definitions, including *set* and *get* 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 }
52 // end function setSecond
53
54 // return hour value
55 int Time::getHour()
56 {
57     return hour;
58 }
59 // end function getHour
60
61 // return minute value
62 int Time::getMinute()
63 {
64     return minute;
65 }
66 // end function getMinute
67

```

Set function performs validity checks before modifying data.

Get functions allow client to read data.



[Outline](#)

71

time3.cpp (3 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

68 // return second value
69 int Time::getSecond()
70 {
71     return second;
72 }
73 // end function getSecond
74
75 // print Time in universal format
76 void Time::printUniversal()
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()
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

```

Get function allows client to read data.



[Outline](#)

72

time3.cpp (4 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 16: **Time** class member-function definitions, including *set* and *get* functions. (part 2 of 2)

```

1 // Fig. 6.20: fig06_20.cpp
2 // Demonstrating the Time class set and get functions
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 // include definition of class Time from time3.h
9 #include "time3.h"
10
11 void incrementMinutes( Time &, const int ); // prototype
12
13 int main()
14 {
15     Time t; // create Time object
16
17     // set time using individual set functions
18     t.setHour( 17 ); // set hour to valid value
19     t.setMinute( 34 ); // set minute to valid value
20     t.setSecond( 25 ); // set second to valid value
21

```



Outline

73

fig06_20.cpp
(1 of 3)

Invoke set functions to set valid values.

© 2003 Prentice Hall, Inc.
All rights reserved.

```

22 // use get functions to obtain hour, minute and second
23 cout << "Result of setting all valid values:\n"
24     << " Hour: " << t.getHour()
25     << " Minute: " << t.getMinute()
26     << " Second: " << t.getSecond();
27
28 // set time using individual set functions
29 t.setHour( 234 ); // invalid hour set to 0
30 t.setMinute( 43 ); // set minute to valid value
31 t.setSecond( 6373 ); // invalid second set to 0
32
33 // display hour, minute and second after setting
34 // invalid hour and second values
35 cout << "\n\nResult of attempting to set invalid hour and"
36     << " second:\n Hour: " << t.getHour()
37     << " Minute: " << t.getMinute()
38     << " Second: " << t.getSecond() << "\n\n";
39
40 t.setTime( 11, 58, 0 ); // set time
41 incrementMinutes( t, 3 ); // increment t's minute by 3
42
43 return 0;
44
45 } // end main
46

```



Outline

74

Attempt to set invalid values using set functions.

Invalid values result in setting data members to 0.

Modify data members using function setTime.

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 17: Set and get functions manipulating an object's private data. (part 1 of 2)

```

47 // add specified number of minutes to a Time object
48 void incrementMinutes( Time &tt, const int count )
49 {
50     cout << "Incrementing minute " << count
51         << " times:\nStart time: ";
52     tt.printStandard();
53
54     for ( int i = 0; i < count; i++ ) {
55         tt.setMinute( ( tt.getMinute() + 1 ) % 60 );
56
57         if ( tt.getMinute() == 0 )
58             tt.setHour( ( tt.getHour() + 1 ) % 24);
59
60         cout << "\nminute + 1: ";
61         tt.printStandard();
62
63     } // end for
64
65     cout << endl;
66
67 } // end function incrementMinutes

```



[Outline](#)

75

fig06_20.cpp

Using get functions to read data and set functions to modify data.

© 2003 Prentice Hall, Inc.
All rights reserved.

```

Result of setting all valid values:
Hour: 17 Minute: 34 Second: 25

Result of attempting to set invalid hour and second:
Hour: 0 Minute: 43 Second: 0

Incrementing minute 3 times:
Start time: 11:58:00 AM
minute + 1: 11:59:00 AM
minute + 1: 12:00:00 PM
minute + 1: 12:01:00 PM

```



[Outline](#)

76

fig06_20.cpp
output (1 of 1)

Attempting to set data members with invalid values results in error message and members set to 0.

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 18: *Set* and *get* functions manipulating an object's **private** data. (part 2 of 2)

7 Default Memberwise Assignment

The assignment operator (=) can be used to assign an object to another object of the same type.

- Assigning objects
 - Assignment operator (=)
 - Can assign one object to another of same type
 - Default: memberwise assignment
 - Each right member assigned individually to left member
- Passing, returning objects
 - Objects passed as function arguments
 - Objects returned from functions
 - Default: pass-by-value
 - * Copy of object passed, returned
 - Copy constructor; Copy original values into new object

Member wise assignment can cause serious problems when used with a class whose data members contain pointers to dynamically allocated storage.

```

1 // Fig. 6.24: fig06_24.cpp
2 // Demonstrating that class objects can be assigned
3 // to each other using default memberwise assignment.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 // class Date definition
10 class Date {
11
12 public:
13     Date( int = 1, int = 1, int = 1990 ); // default constructor
14     void print();
15
16 private:
17     int month;
18     int day;
19     int year;
20
21 }; // end class Date
22

```



[Outline](#)

84

fig06_24.cpp
(1 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

23 // Date constructor with no range checking
24 Date::Date( int m, int d, int y )
25 {
26     month = m;
27     day = d;
28     year = y;
29
30 } // end Date constructor
31
32 // print Date in the format mm-dd-yyyy
33 void Date::print()
34 {
35     cout << month << "-" << day << "-" << year;
36
37 } // end function print
38
39 int main()
40 {
41     Date date1( 7, 4, 2002 );
42     Date date2; // date2 defaults to 1/1/1990
43

```



[Outline](#)

85

fig06_24.cpp
(2 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 19: Default memberwise assignment. (part 1 of 2)


```
44  cout << "date1 = ";
45  date1.print();
46  cout << "\ndate2 = ";
47  date2.print();
48
49  date2 = date1; // default memberwise assignment
50
51  cout << "\n\nAfter default memberwise assignment, date2 = ";
52  date2.print();
53  cout << endl;
54
55  return 0;
56
57 } // end main

date1 = 7-4-2002
date2 = 1-1-1990

After default memberwise assignment, date2 = 7-4-2002
```

Default memberwise assignment assigns each member of `date1` individually to each member of `date2`.



Outline

86

fig06_24.cpp
(3 of 3)

fig06_24.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 20: Default memberwise assignment. (part 2 of 2)

8 Software Reusability

- Class libraries
 - Well-defined
 - Carefully tested
 - Well-documented
 - Portable
 - Widely available
- Speeds development of powerful, high-quality software
 - Rapid applications development (RAD)
- Resulting problems
 - Cataloging schemes
 - Licensing schemes
 - Protection mechanisms