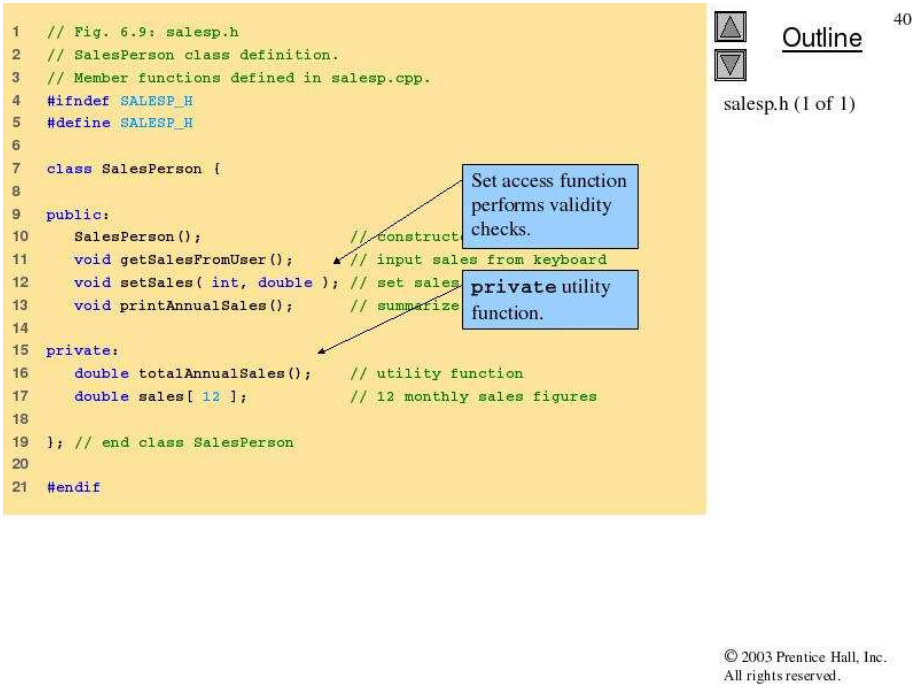


Figure 1: **SalesPerson** class definition

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 demonstarates 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)


```
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 in sales array
41 void SalesPerson::setSales( int month, doub
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;
```

Set access function performs validity checks.

Outline

42

salesp.cpp (2 of 3)

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

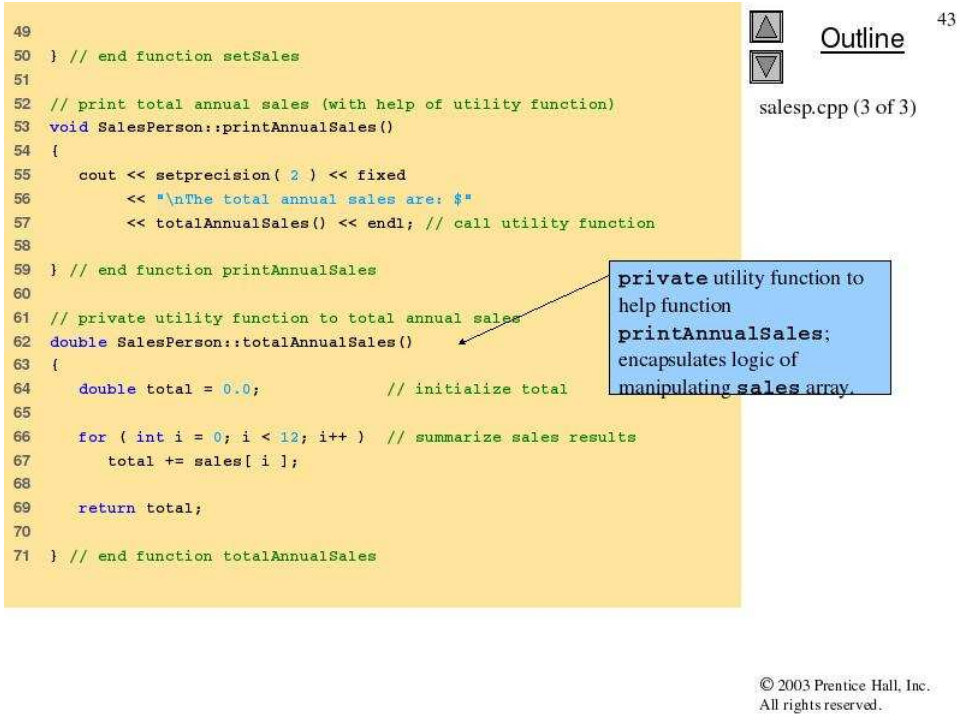
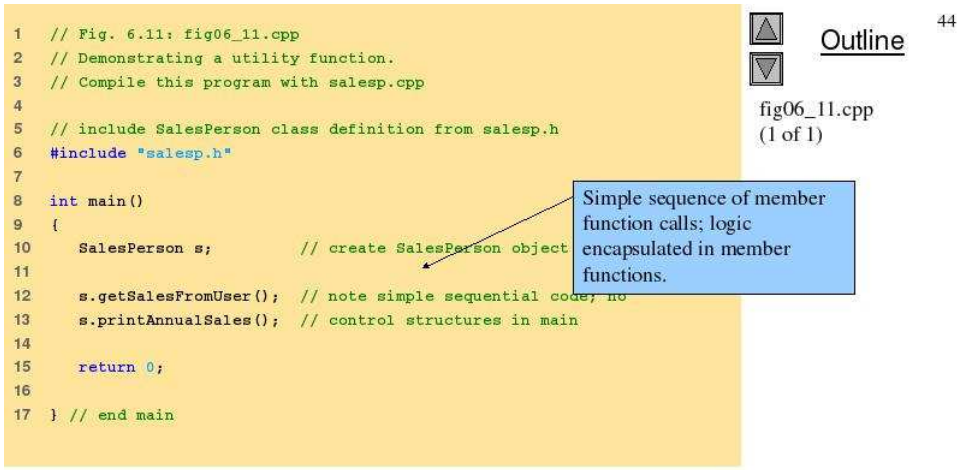


Figure 3: **SalesPerson** class member-function definitions (part 2 of 2)



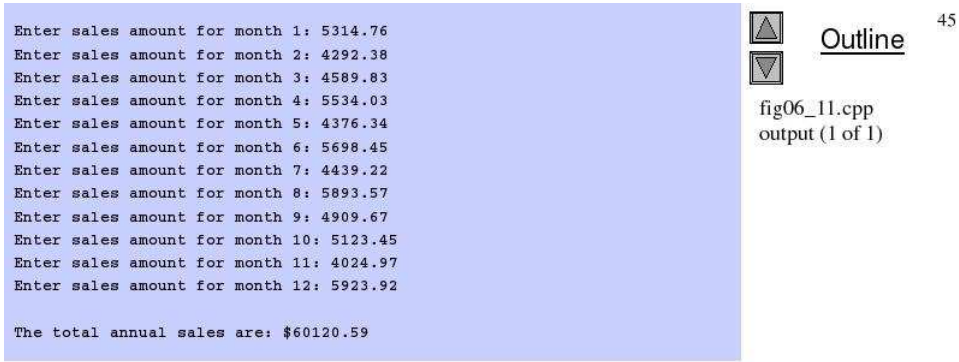


Figure 4: Utility function demonstration

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

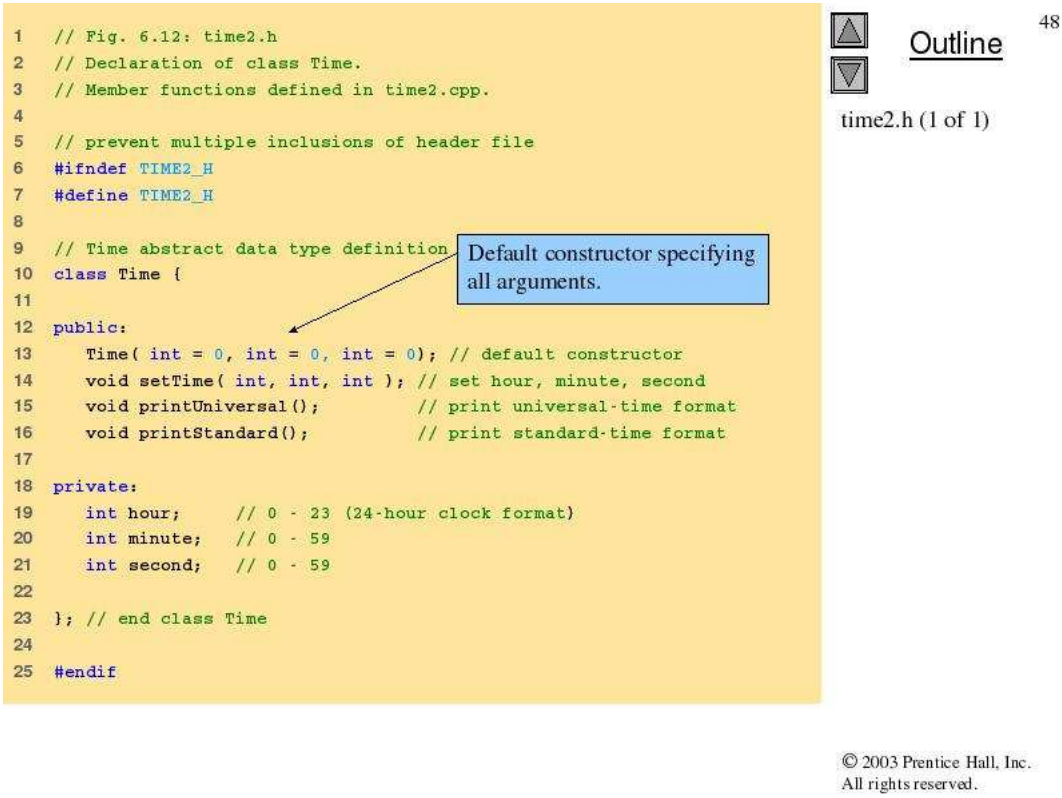
    ```
    Class-type ObjectName( value1,value2,...};
    ```

  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
```


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
```


time2.cpp (1 of 3)

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


```
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
```


time2.cpp (2 of 3)

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

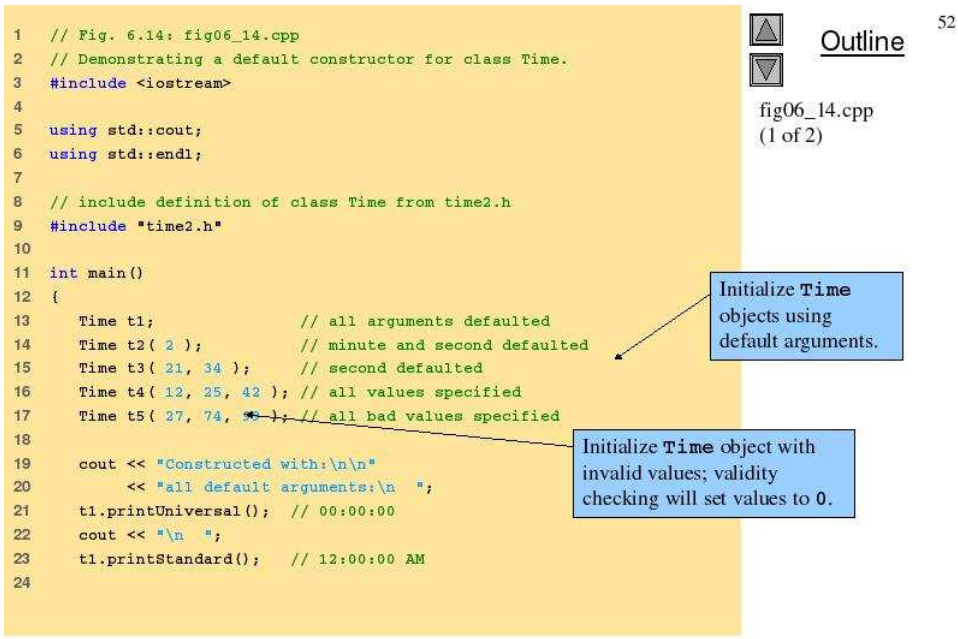
```
42 // print Time in standard format
43 void Time::printStandard()
44 {
45    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
46         << ":" << setfill( '0' ) << setw( 2 ) << minute
47         << ":" << setw( 2 ) << second
48         << ( hour < 12 ? " AM" : " PM" );
49
50 } // end function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

Outline

51

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, 99 ); // 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
```


```
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
```

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

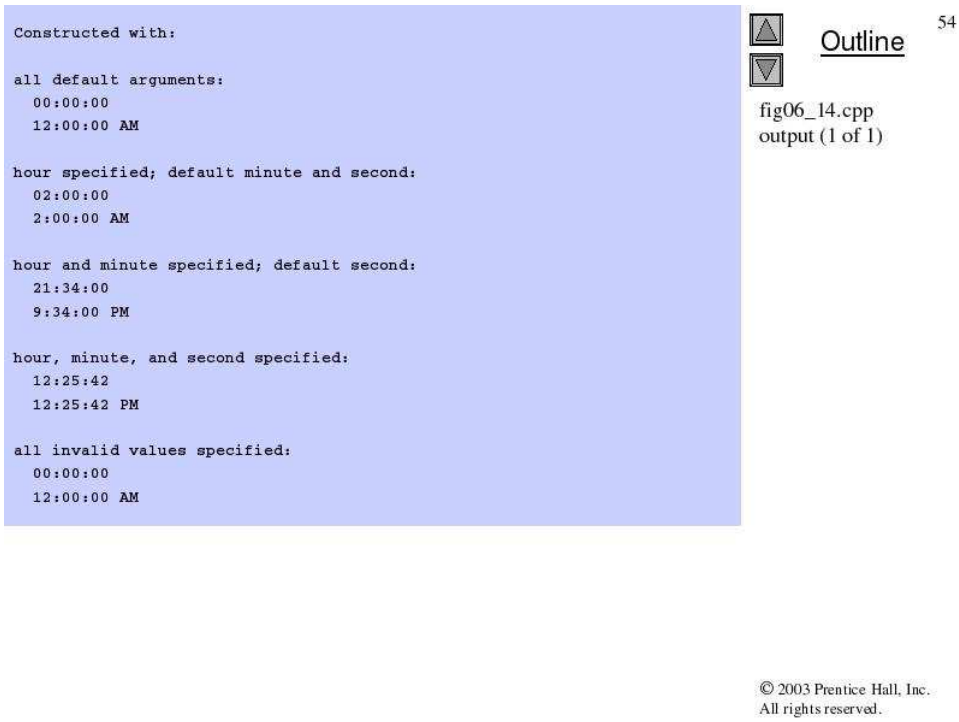


Figure 9: Constructor with default arguments. (part 2 of 2)

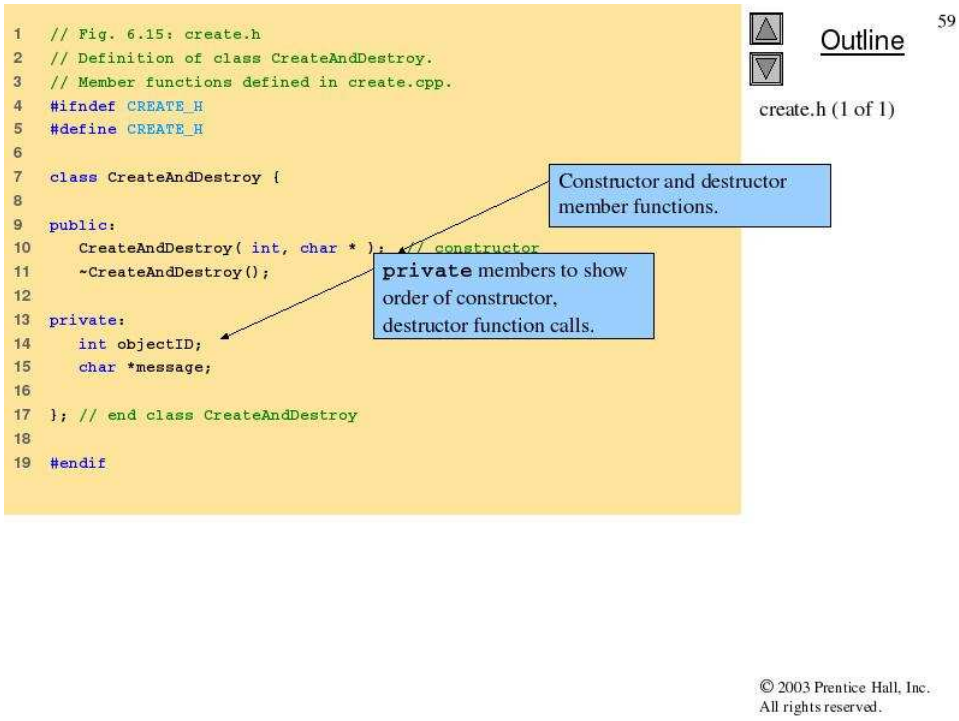
4 Destructors

- 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

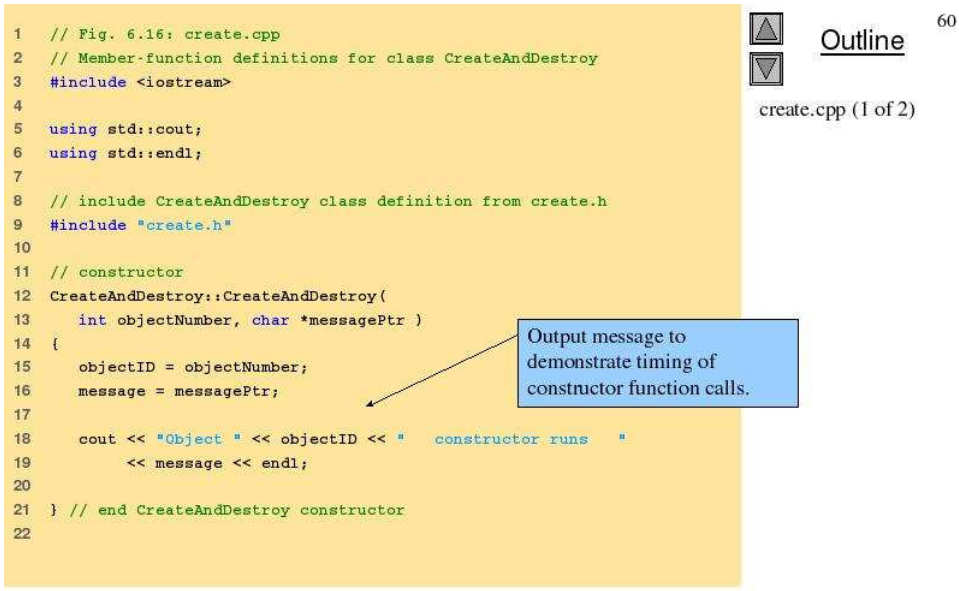
- Constructors and destructors; Called implicitly by compiler
- Order of function calls
  - Depends on order of execution; When execution enters and exits scope of objects
  - Generally, destructor calls reverse order of constructor calls
- Order of constructor, destructor function calls
  - Global scope 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
```

Figure 10: **CreateAndDestroy** class definition.



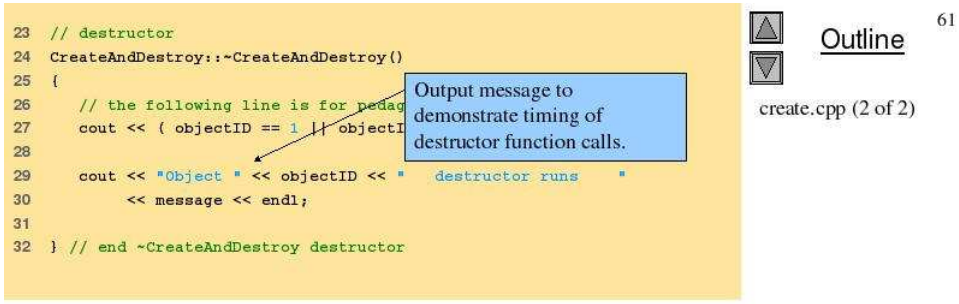


Figure 11: **CreateAndDestroy** class member-function definitions.

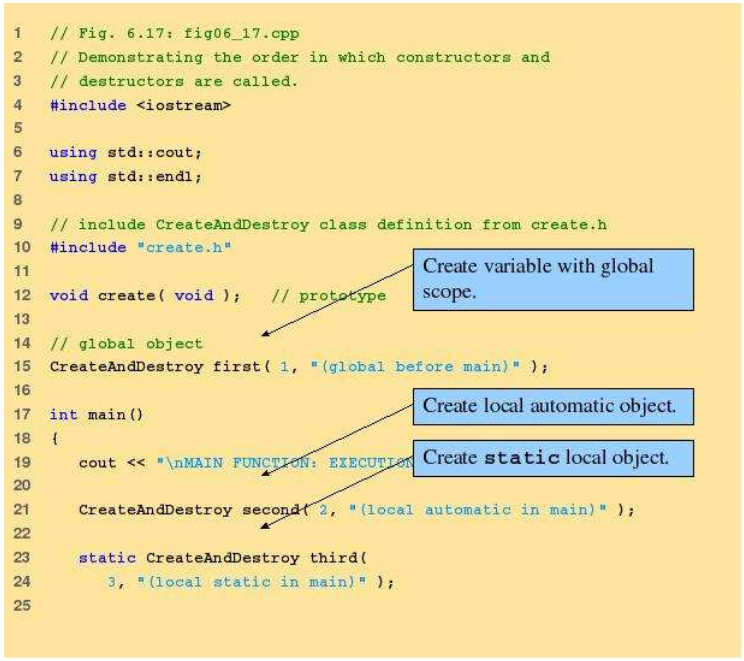


fig06_17.cpp
(1 of 3)

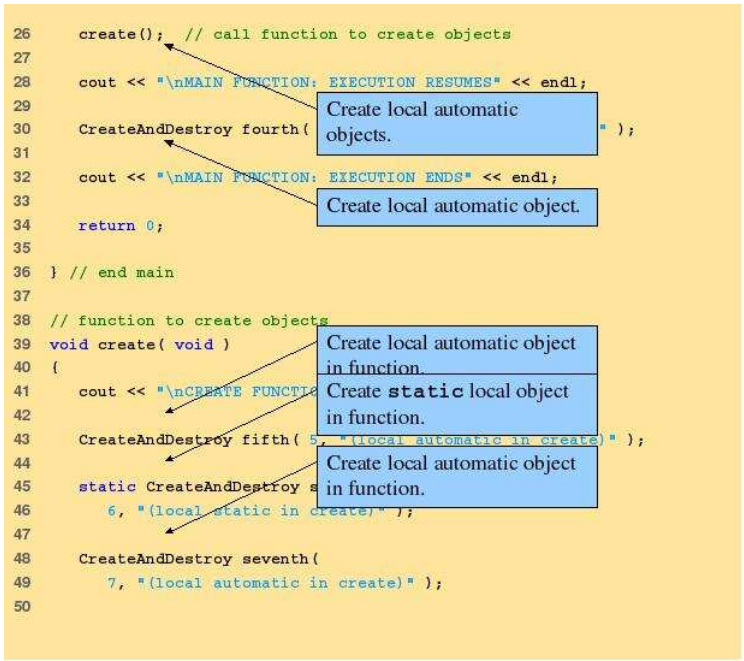


fig06_17.cpp
(2 of 3)

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

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

Outline

64

fig06_17.cpp
(3 of 3)


```
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)

Global object constructed

Local **static** object constructed on first function call and destroyed after **main** execution ends.

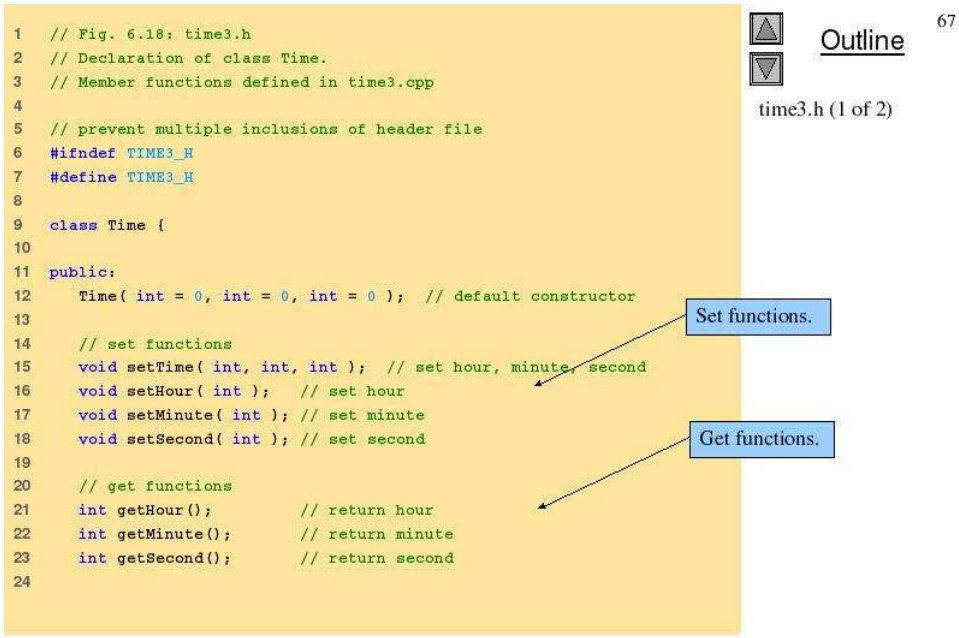
Figure 13: Order in which constructors and destructors are called. (part 2 of 2)

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* (,.e., read) the values of **private** data members. These functions need not be called *set* and *get* specifically, but they often are.

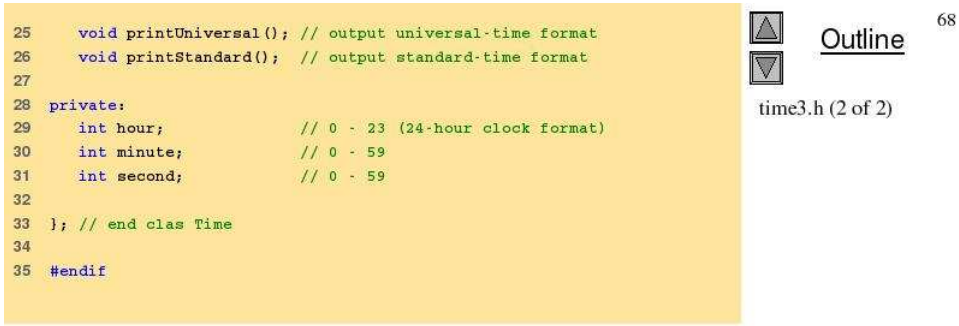
- Set functions
  - Perform validity checks before modifying **private** data
  - Notify if invalid values
  - Indicate with return values
- Get functions
  - "Query" functions
  - Control format of data returned

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



Outline

time3.h (1 of 2)



Outline

time3.h (2 of 2)

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)


```
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
```

Call set functions to perform validity checking.

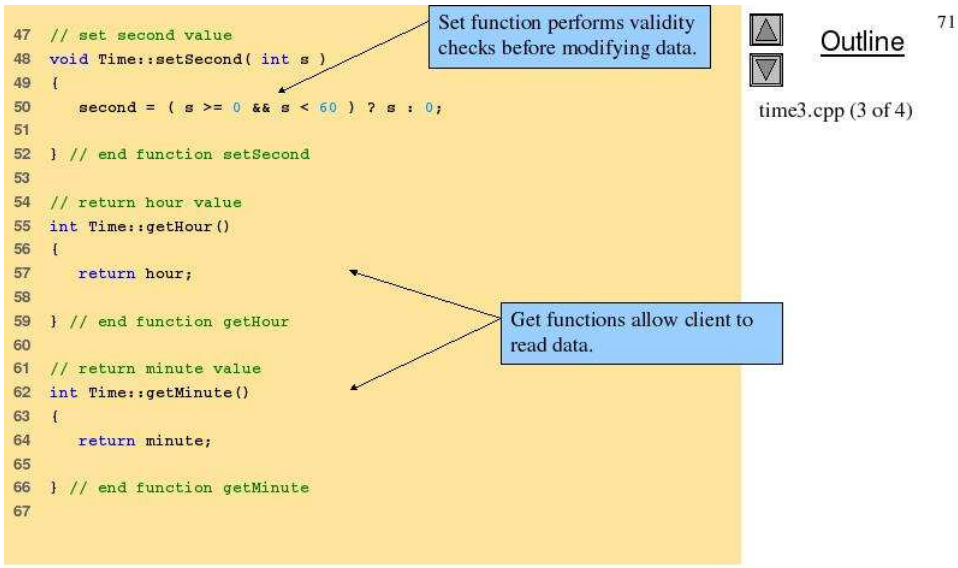
Set functions perform validity checks before modifying data.

Outline

70

time3.cpp (2 of 4)

Figure 15: **Time** class member-function definitions,including *set* and *get* functions. (part 1 of 2)



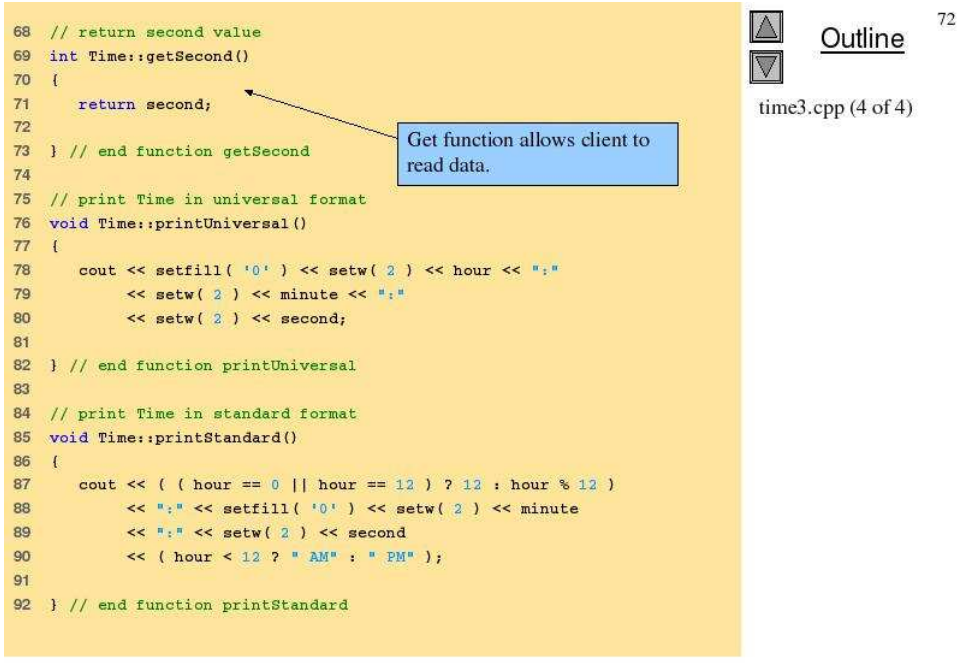
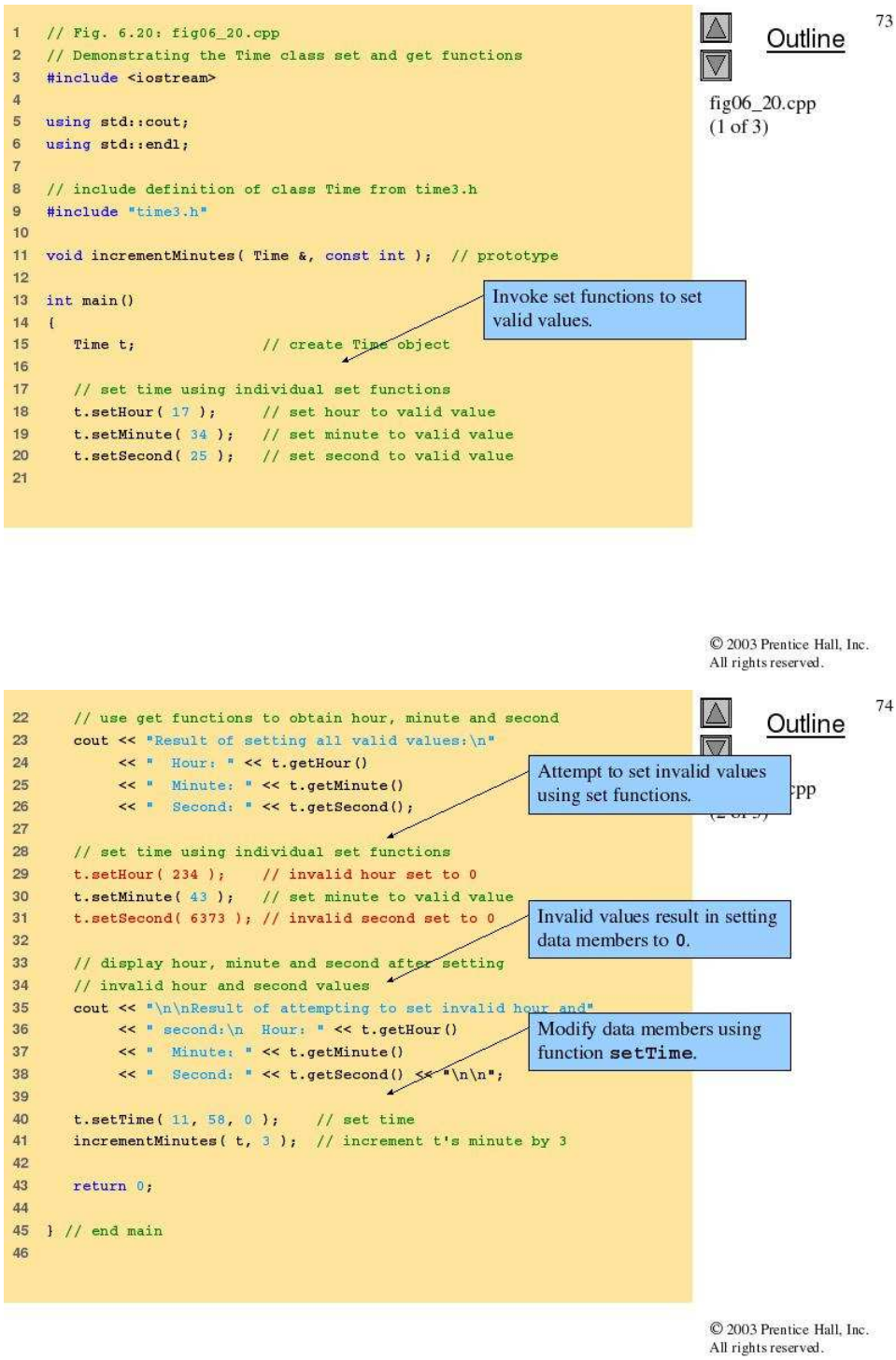
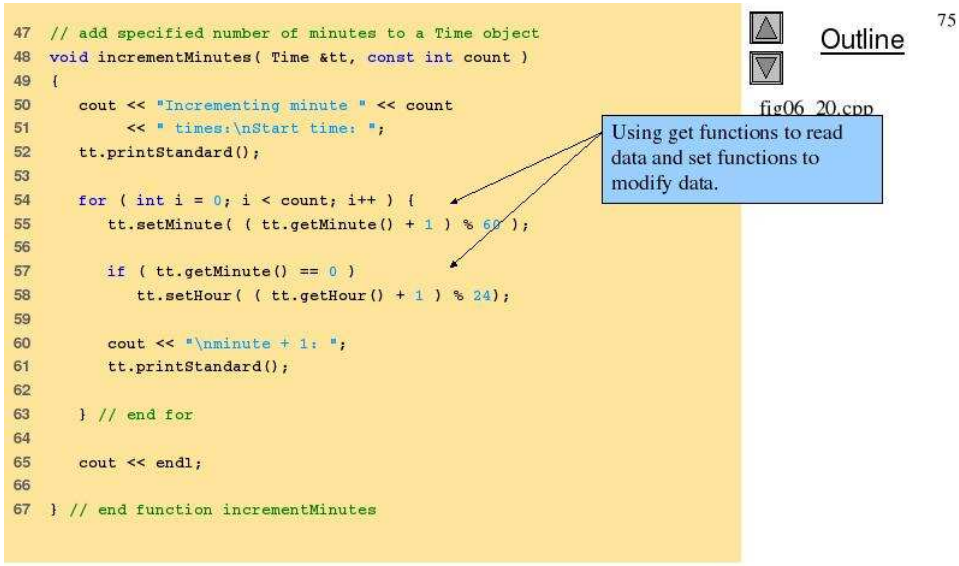


Figure 16: **Time** class member-function definitions,including *set* and *get* functions. (part 2 of 2)




```
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
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
```

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



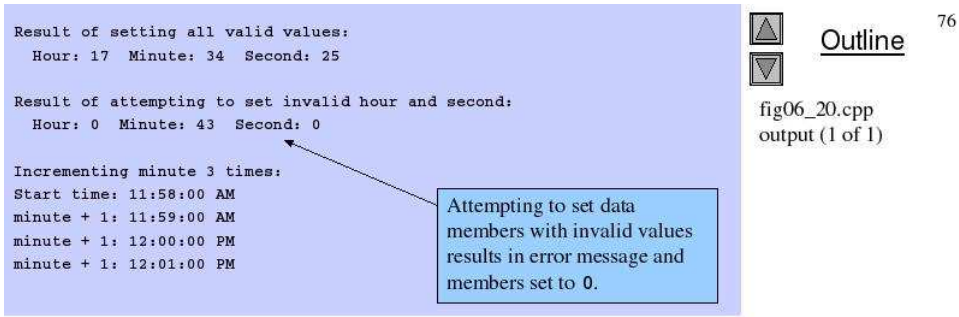


Figure 18: *Set* and *get* functions manipulating an object's **private** data. (part 2 of 2)

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)


```
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)

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

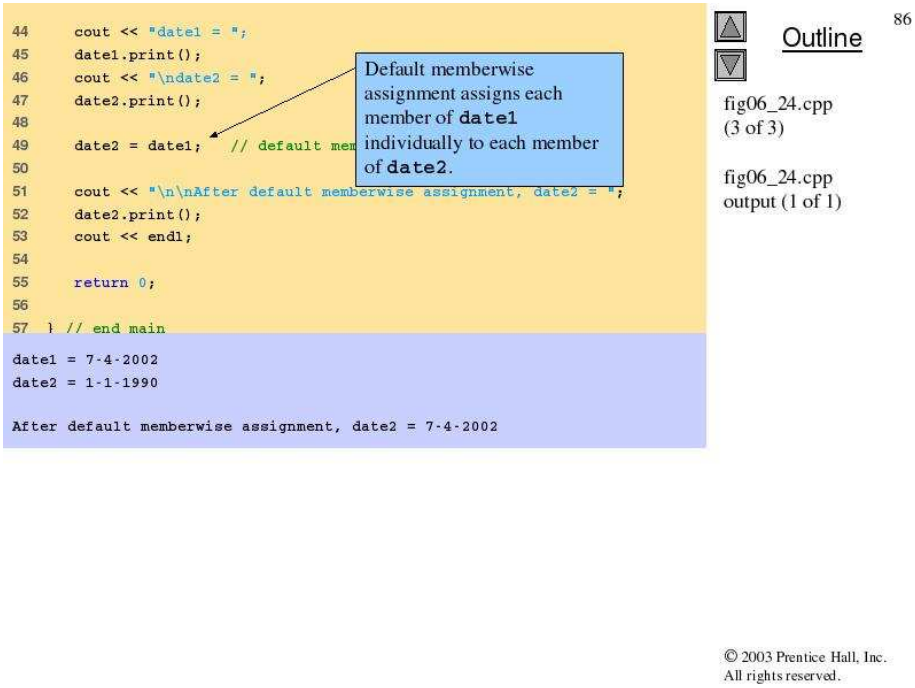


Figure 20: Default memberwise assignment. (part 2 of 2)

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