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.

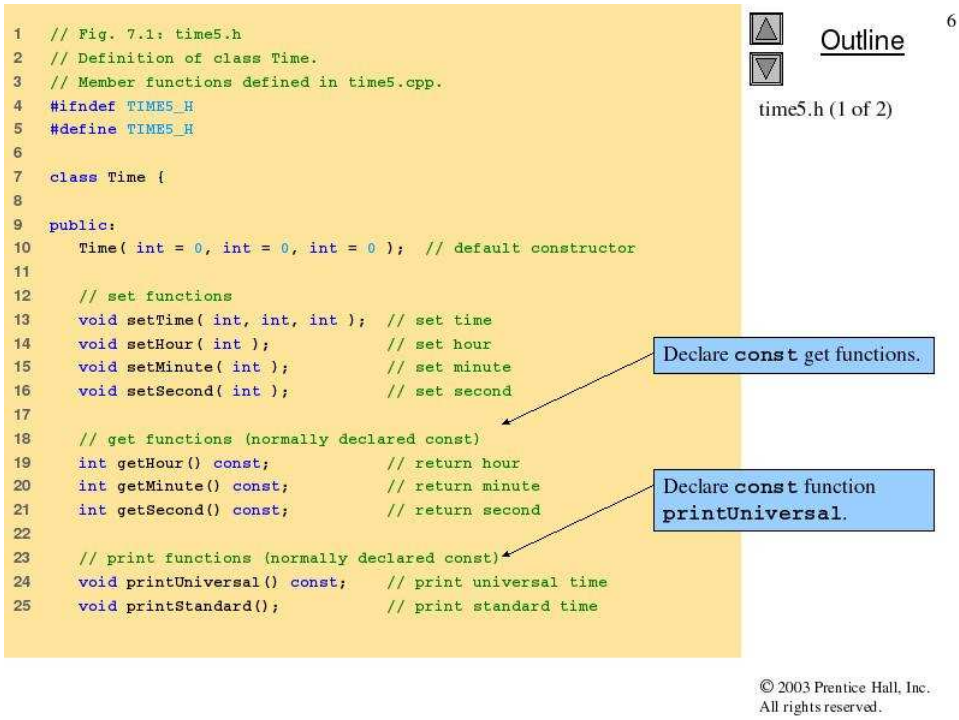
- Principle of least privilege; Only allow modification of necessary objects
- Keyword const
  - Specify object not modifiable
  - Compiler error if attempt to modify **const** object
  - Example
    - **const Time noon(12, 0, 0);**
    - Declares **const** object **noon** of class **Time**
    - Initializes 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.



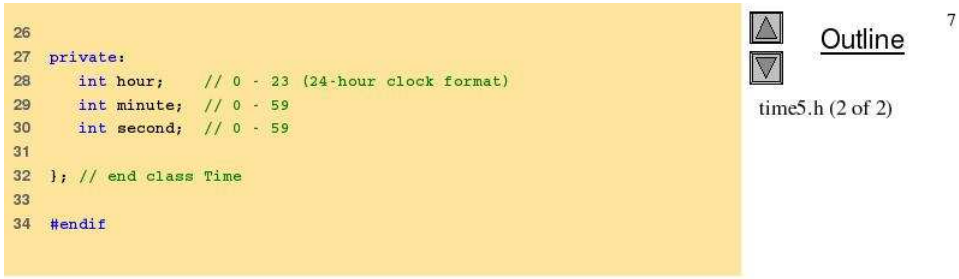


Figure 1: **Time** class definition with **const** member functions.

```
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)

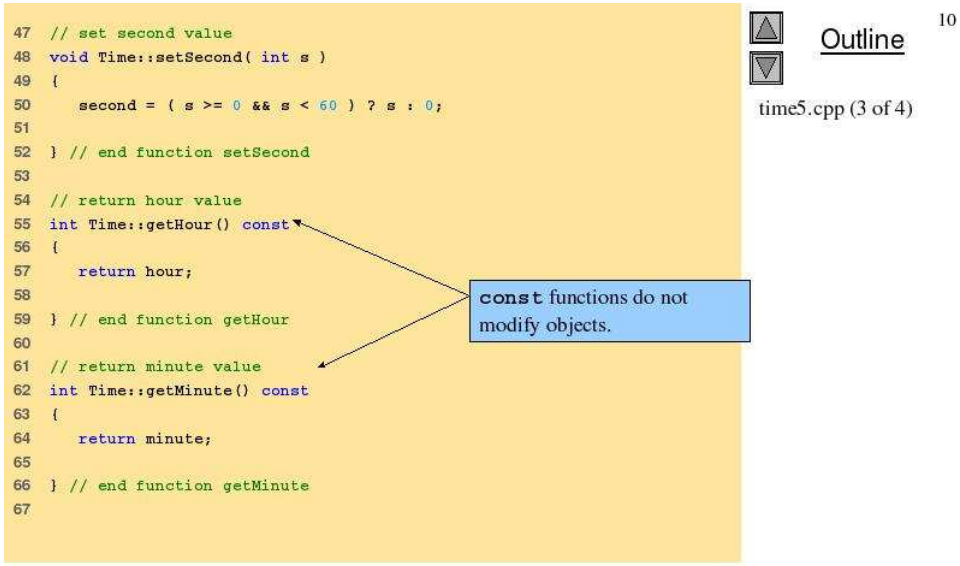

```
24 // set hour, minute and second values
25 void Time::setTime( int hour, int minute, int second )
26 {
27    setHour( hour );
28    setMinute( minute );
29    setSecond( second );
30
31 } // end function setTime
32
33 // set hour value
34 void Time::setHour( int h )
35 {
36    hour = ( h >= 0 && 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)

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



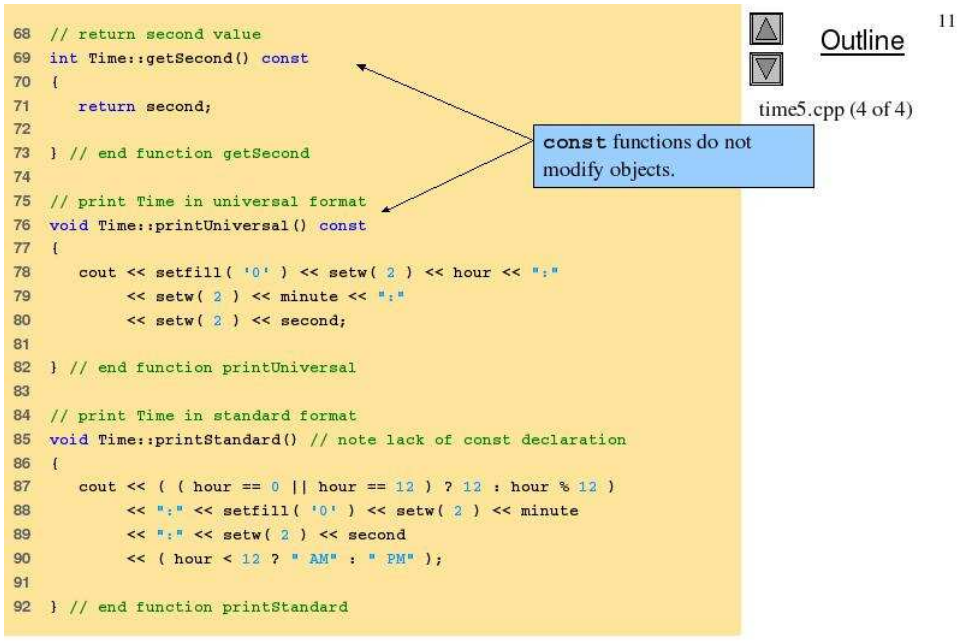
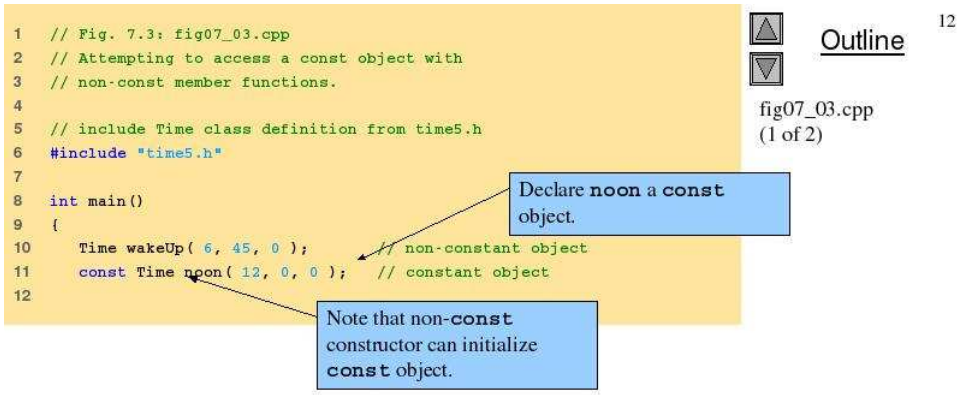


Figure 3: **Time** class member-function definitions, including **const** member functions. (part 2 of 2)



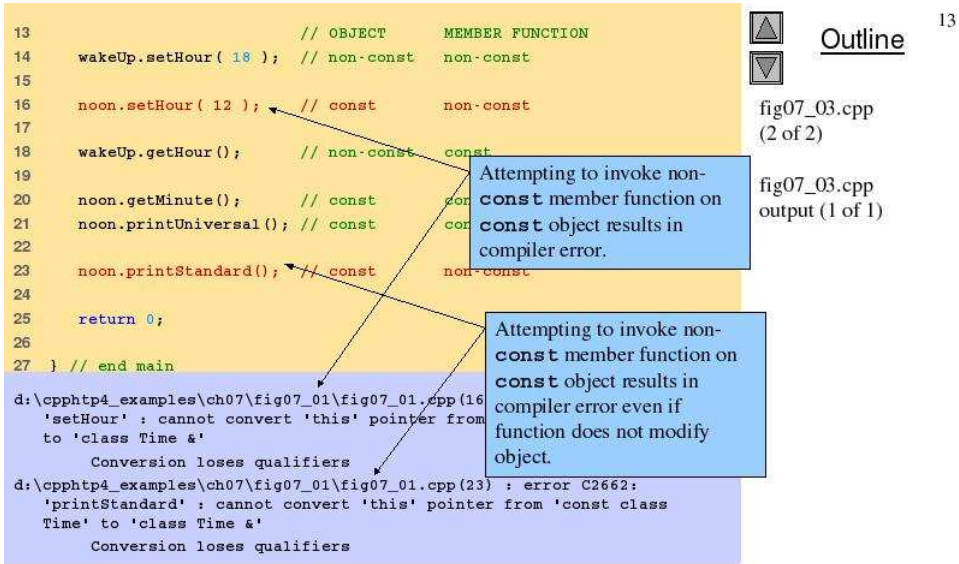


Figure 4: **const** objects and **const** member functions.

```
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)


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

Member initializer list separated by colon.

ncrement as **const**

Member initializer syntax can be

Member initializer syntax must be used for **const** data member **increment**.

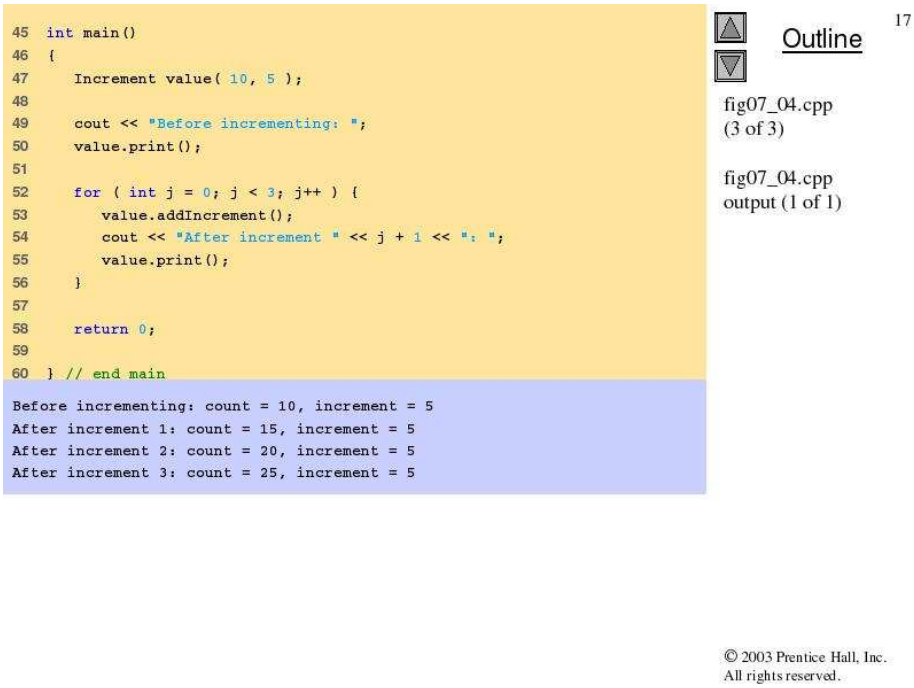
Member initializer consists of data member name (**increment**) followed by parentheses containing initial value (**c**).

Outline

16

fig07_04.cpp
(2 of 3)

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

fig07_04.cpp
(3 of 3)

fig07_04.cpp
output (1 of 1)

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)


```
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 {              // Constant member
31    count = c;       // allowed becs
32    increment = i;  // ERROR: Cannot modify a const object
33
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)

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

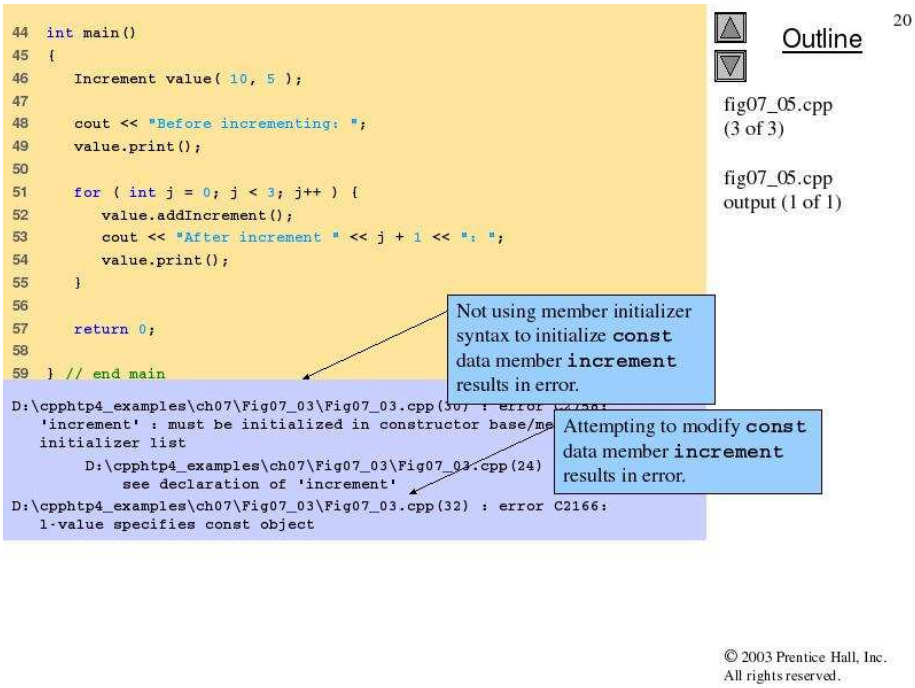


Figure 8: Erroneous attempt to initialize a constant of a built-in data type by assignment. (part 2 of 2)

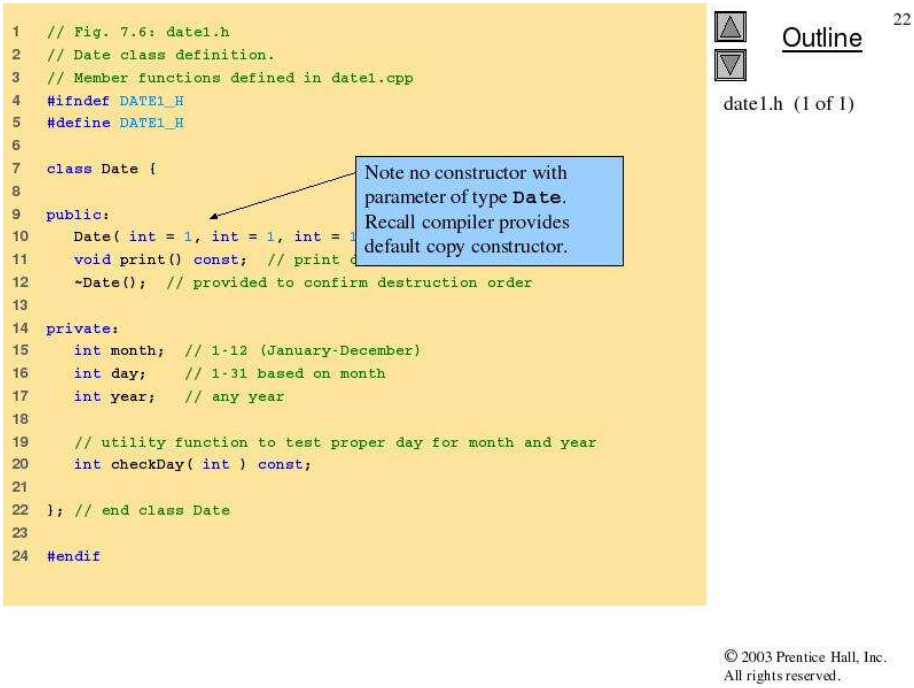


Figure 9: **Date** class definition.

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.

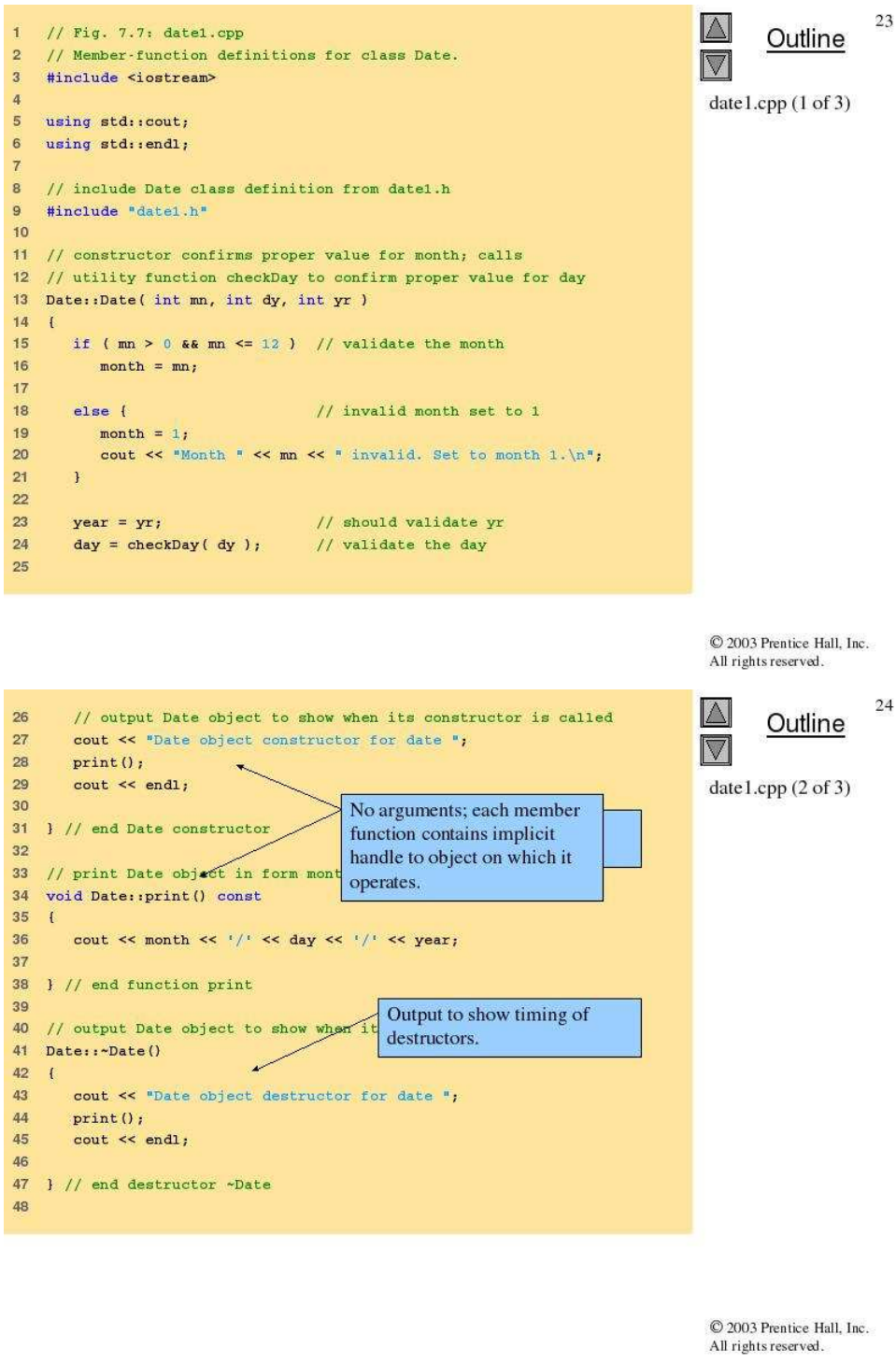
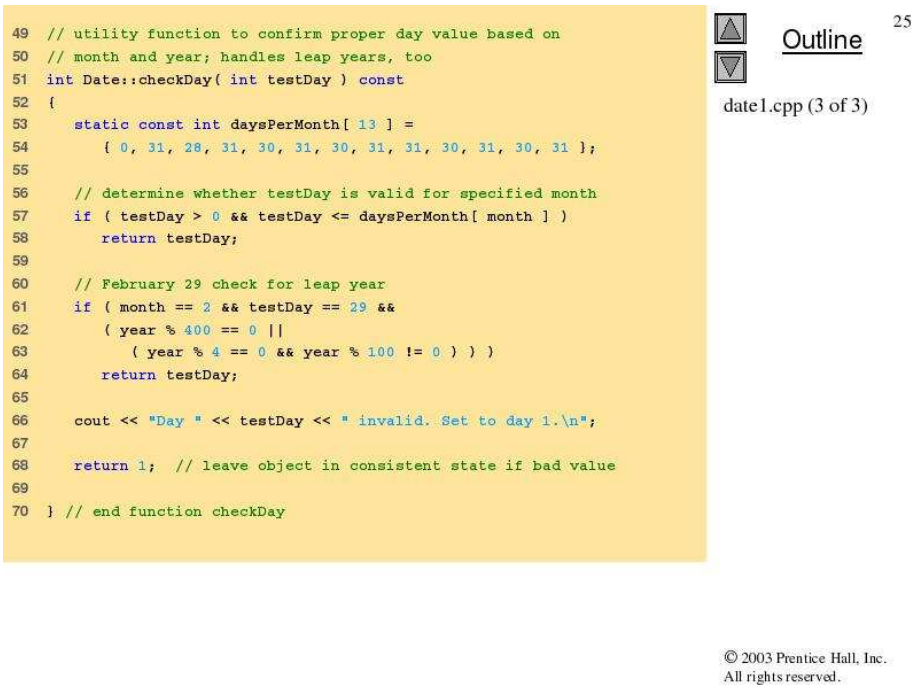
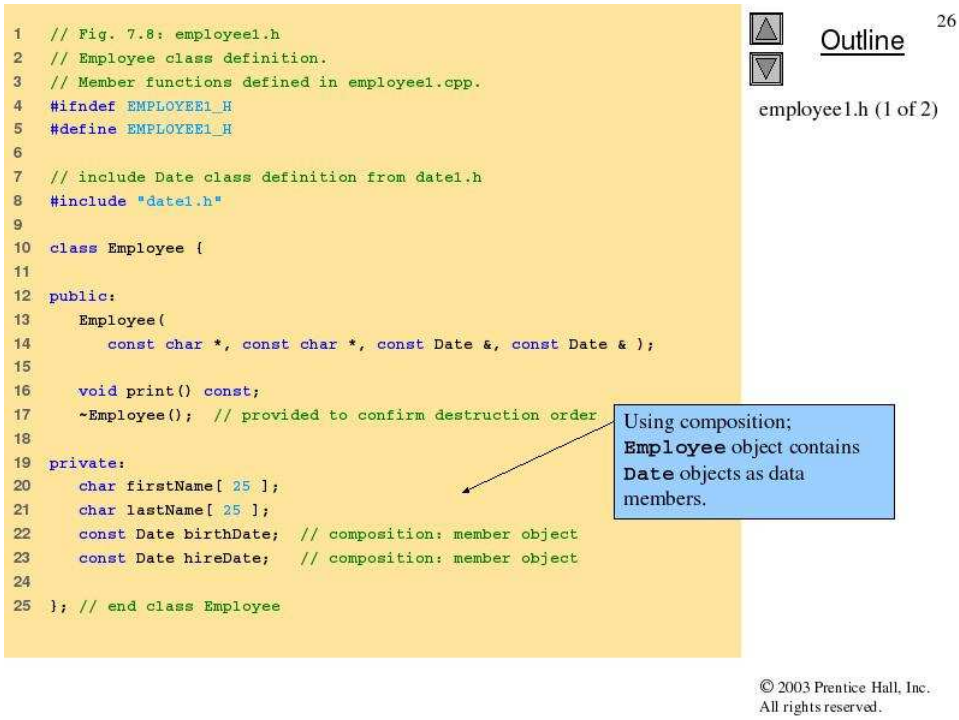


Figure 10: **Date** class member-function definitions. (part 1 of 2)



```
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
```


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



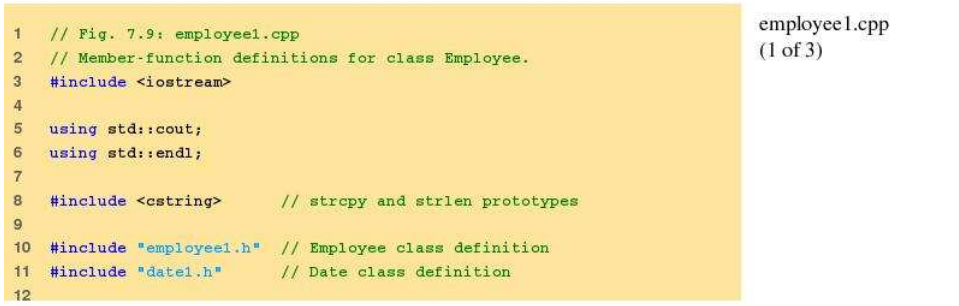


Figure 12: **Employee** class definition showing composition.

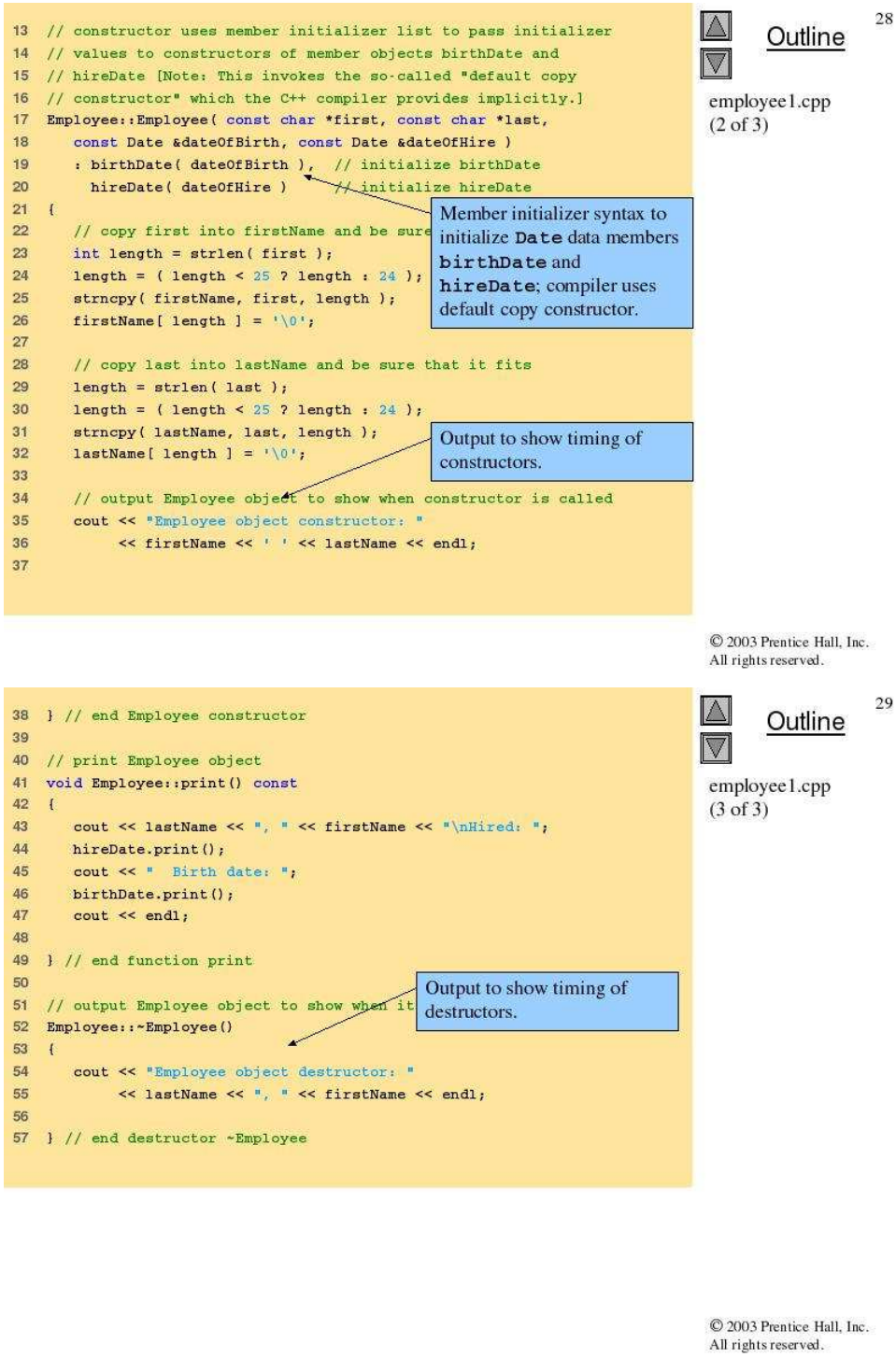
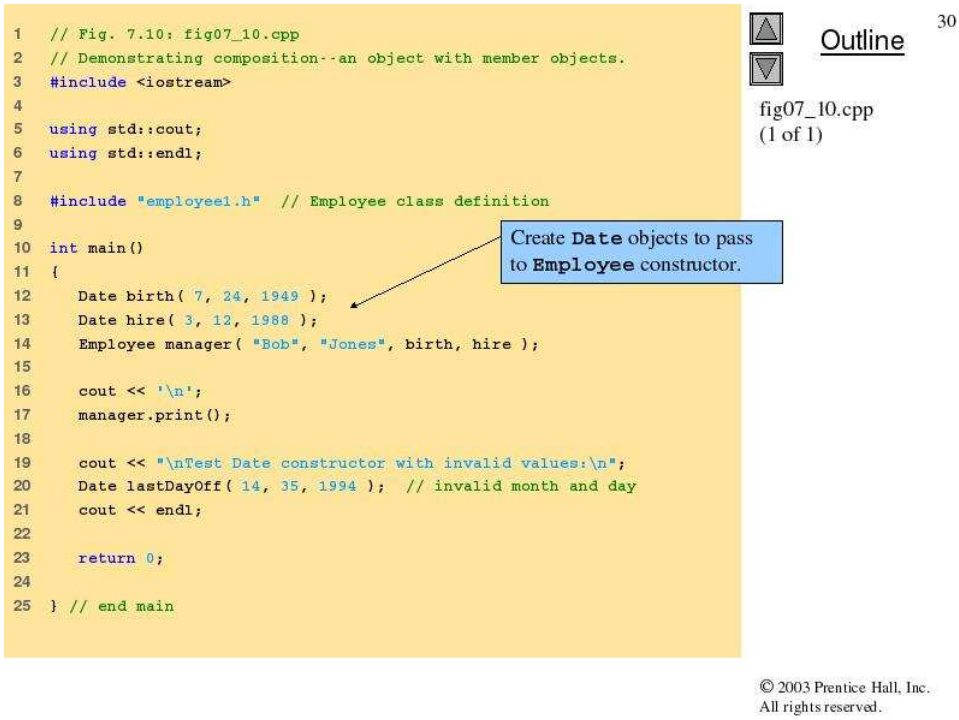


Figure 13: **Employee** class member-function definitions,including constructor with a member-initializer list.



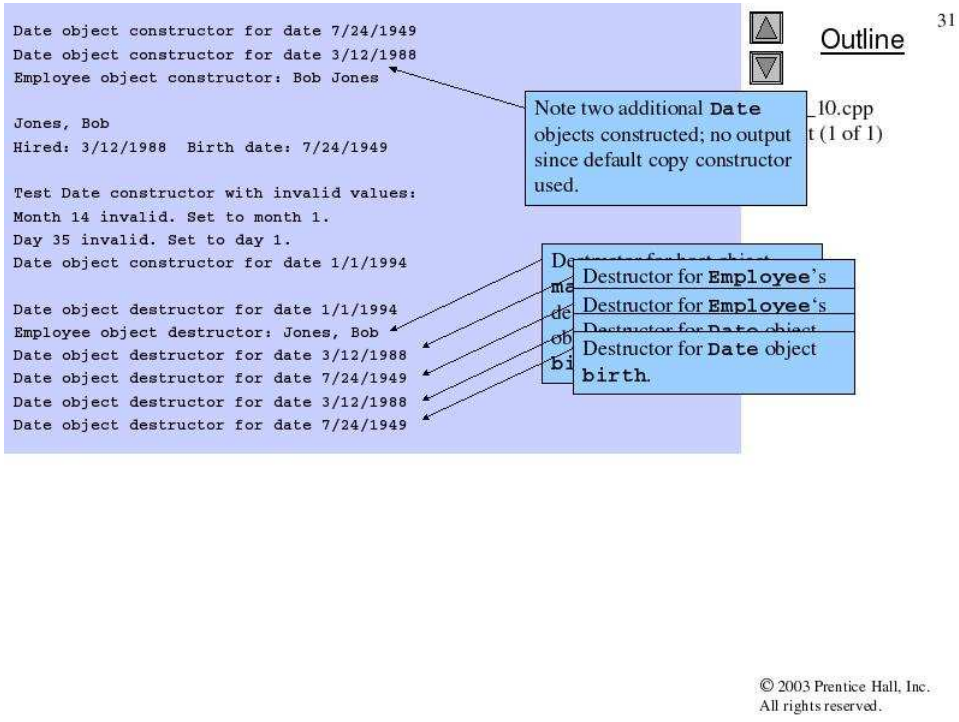
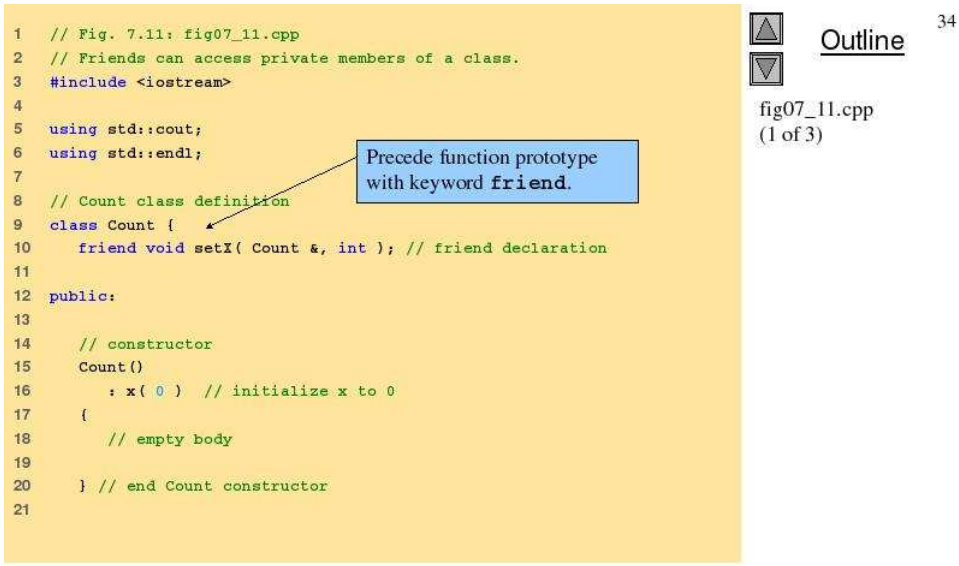


Figure 14: Member-object initializers.

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 **friend**s 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 **cannotSetX** is called to modify **private** data member **x**.



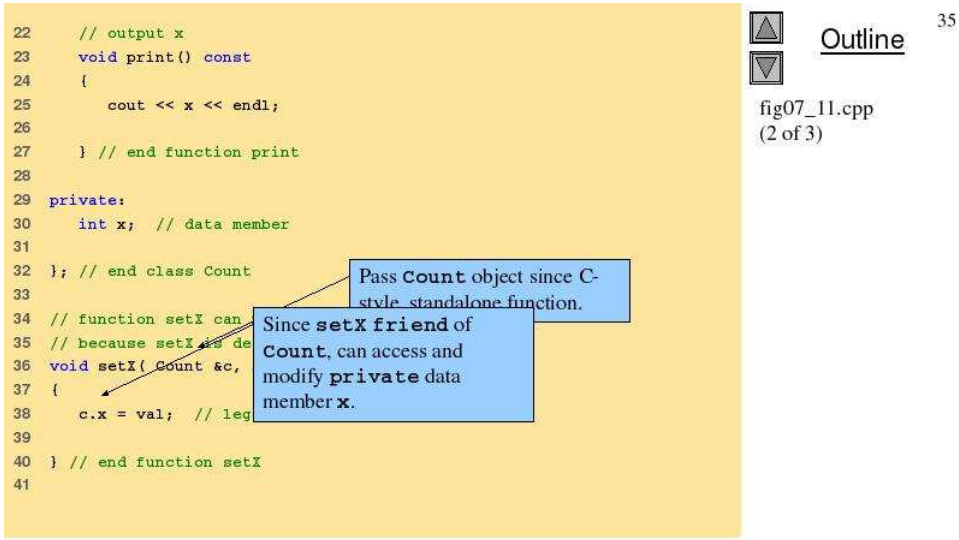


Figure 15: Friends can access **private** members of the class.

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

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

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

Outline

36

fig07_11.cpp
(3 of 3)

fig07_11.cpp
output (1 of 1)


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

Outline

37

fig07_12.cpp
(1 of 3)

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

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
```

Outline

41

fig07_13.cpp
(1 of 3)


```
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 mem
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.

Outline

42

07_13.cpp
of 3)

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

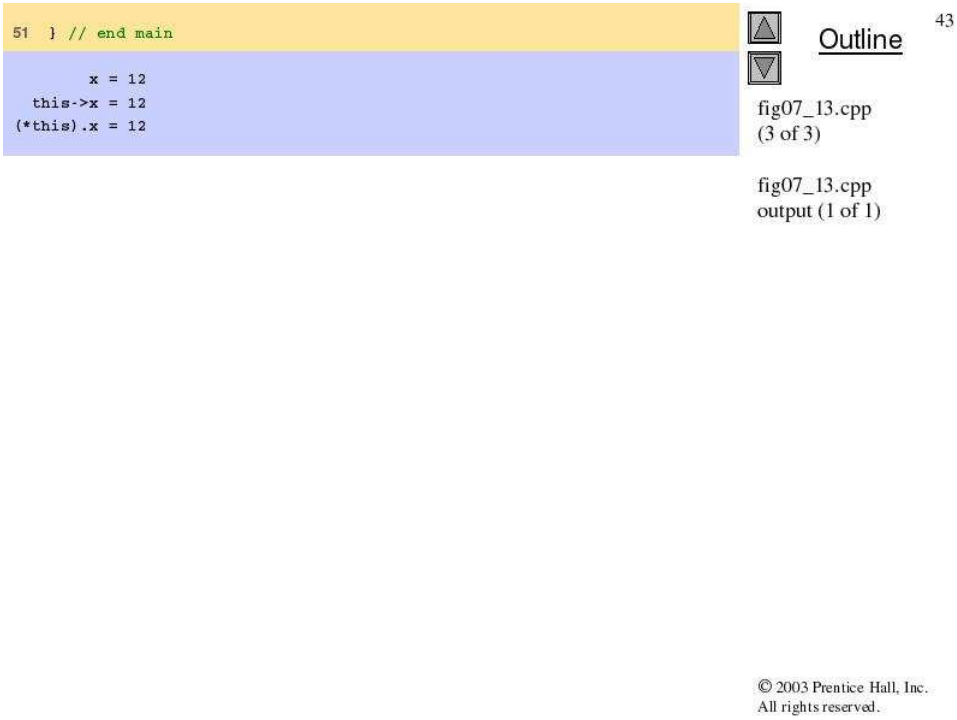


Figure 19: **this** pointer implicitly and explicitly used to access an object's members. (part 2 of 2)

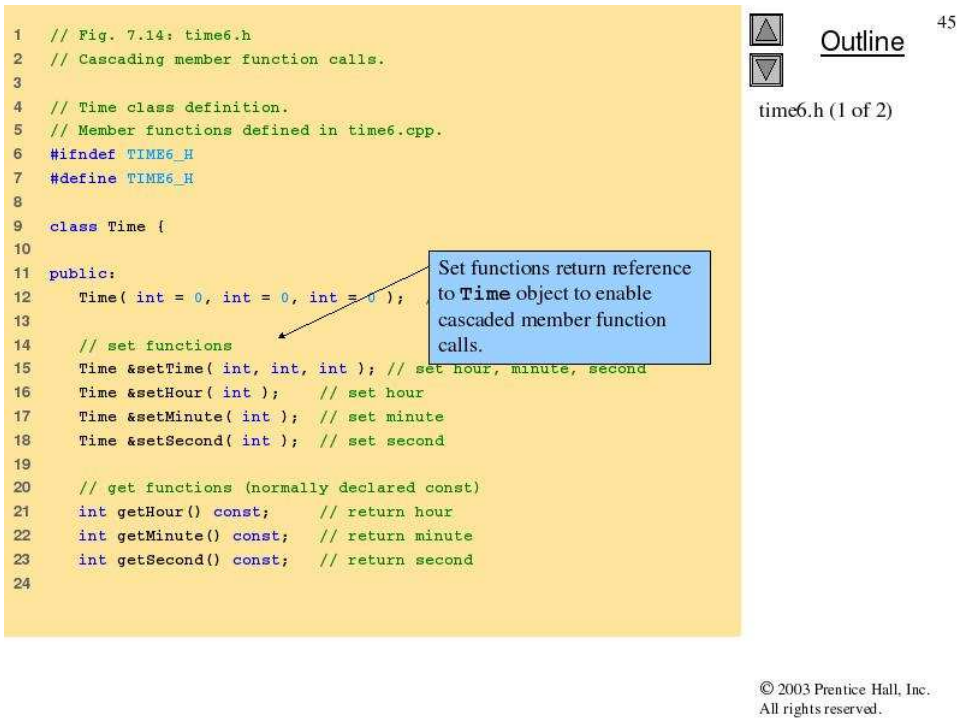


Figure 20: **Time** class definition modified to enable cascaded member-function calls.

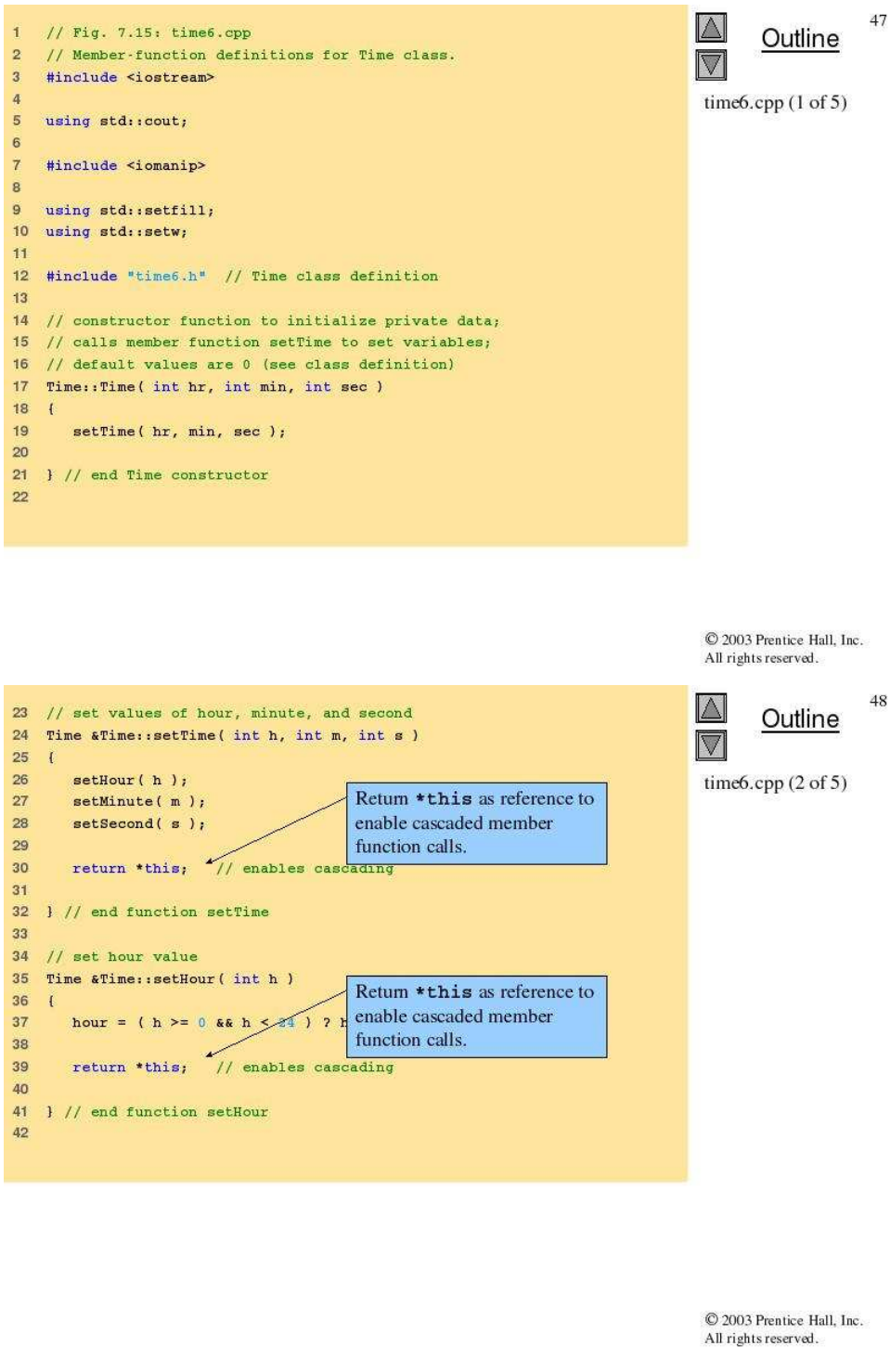
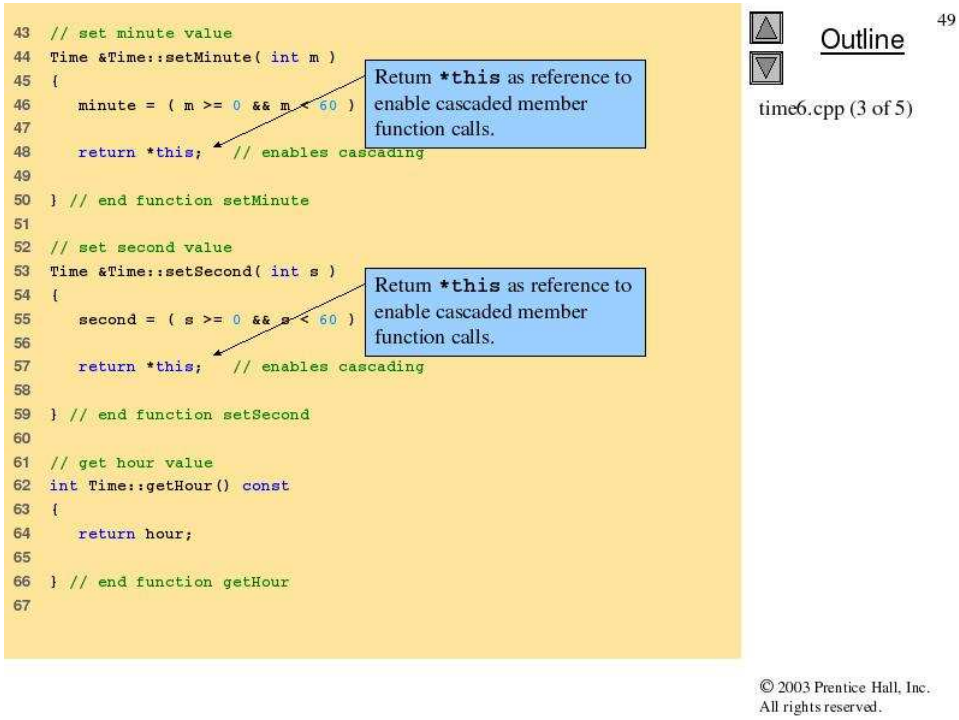


Figure 21: **Time** class member-function definitions modified to enable cascaded member-function calls. (part 1 of 3)



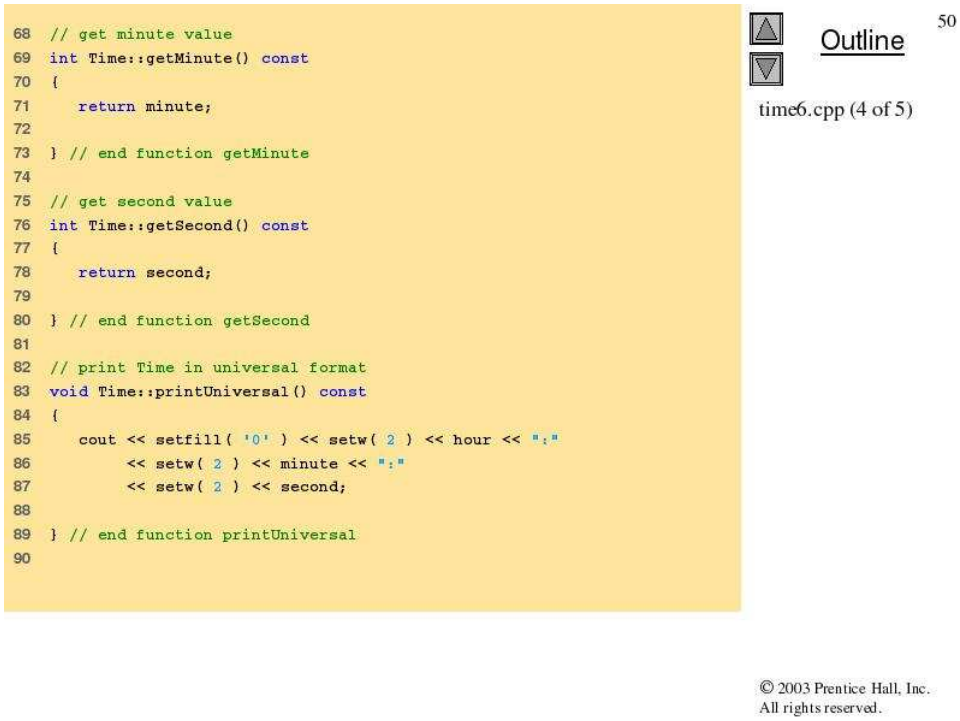
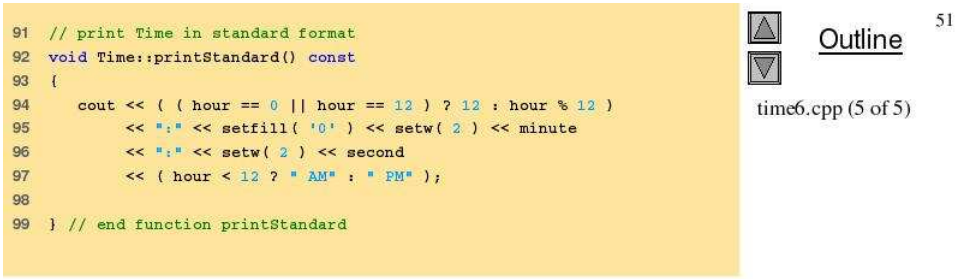
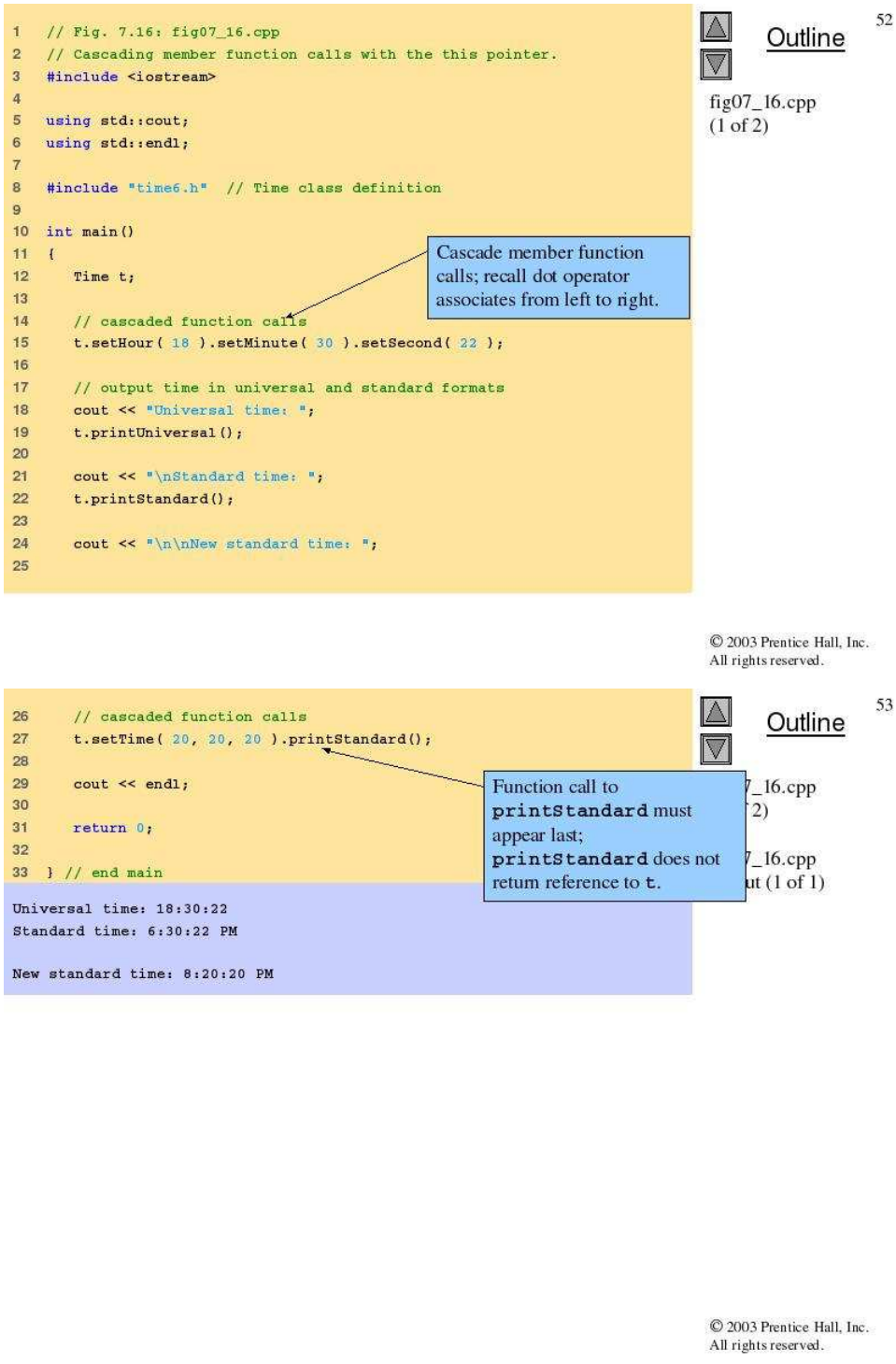


Figure 22: **Time** class member-function definitions modified to enable cascaded member-function calls. (part 2 of 3)



```
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
```


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
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
```

Figure 24: Cascading member-function calls.