1 Operator Overloading

1.1 Introduction

Manipulations on objects were accomplished by sending messages (in the form of member-function calls) to the object.

- Use operators with objects (operator overloading)
  - Clearer than function calls for certain classes
  - Operator sensitive to context
- Examples
  - <<; Stream insertion, bitwise left-shift
  - +; Performs arithmetic on multiple types (integers, floats, etc.)

1.2 Fundamentals of Operator Overloading

C++ programming is a type-sensitive and type-focused process. Operators provide programmers with a concise notation for expressing manipulations of objects of built-in types.

- Types
  - Built in (**int**, **char**) or user-defined
  - Can use existing operators with user-defined types; Cannot create new operators
- Overloading operators
  - Create a function for the class
  - Name function **operator** followed by symbol; **Operator+** for the addition operator +
- Using operators on a class object
  - It must be overloaded for that class
    - Exceptions:
    - Assignment operator, =; Memberwise assignment between objects
    - Address operator, &; Returns address of object

        * Both can be overloaded

- Overloading provides concise notation
    - **object2 = object1.add(object2);**
    - **object2 = object2 + object1;**

Overloading is especially appropriate for mathematical classes. These often require that a substantial set of operators be overloaded to ensure consistency with the way these mathematical classes are handled in the real world. Operator overloading is not automatic, however; the programmer must write operator-overloading functions to perform the desired operations. Sometimes these functions are best made member functions; sometimes they are best as **friend** functions; occasionally the can be made non-member, non-**friend** functions.

1.3 Restrictions on Operator Overloading

Most of C++'s operators can be overloaded.

- Cannot change
    - How operators act on built-in data types; i.e., cannot change integer addition
    - Precedence of operator (order of evaluation); Use parentheses to force order-of-operations
    - Associativity (left-to-right or right-to-left)
    - Number of operands; **&** is unitary, only acts on one operand
- Cannot create new operators
- Operators must be overloaded explicitly; Overloading + does not overload +=

1.4 Operator Functions As Class Members Vs. As Friend Functions

- Operator functions
    - Member functions
        * Use **this** keyword to implicitly get argument

    - Gets left operand for binary operators (like +)
    - Leftmost object must be of same class as operator
  - Non member functions
    - Need parameters for both operands
    - Can have object of different class than operator
    - Must be a **friend** to access **private** or **protected** data
  - Called when
    - Left operand of binary operator of same class
    - Single operand of unitary operator of same class
- Overloaded << operator
  - Left operand of type **ostream &**; Such as **cout** object in **cout << classObject**
  - Similarly, overloaded >> needs **istream &**
  - Thus, both must be non-member functions
- Commutative operators
  - May want + to be commutative; So both "**a + b**" and "**b + a**" work
  - Suppose we have two different classes
  - Overloaded operator can only be member function when its class is on left
    - **HugeIntClass + Long int**
    - Can be member function
  - When other way, need a non-member overload function; **Long int + HugeIntClass**

1.5 Overloading Stream-Insertion and Stream-Extraction Operators

- << and >>
  - Already overloaded to process each built-in type
  - Can also process a user-defined class
- Example program

- Class **PhoneNumber**; Holds a telephone number
- Print out formatted number automatically; **(123) 456-7890**

The program of Figs. 1-2 demonstrates overloading the stream-extraction and stream-insertion operators to handle data of a user-defined telephone number class called **PhoneNumber**.

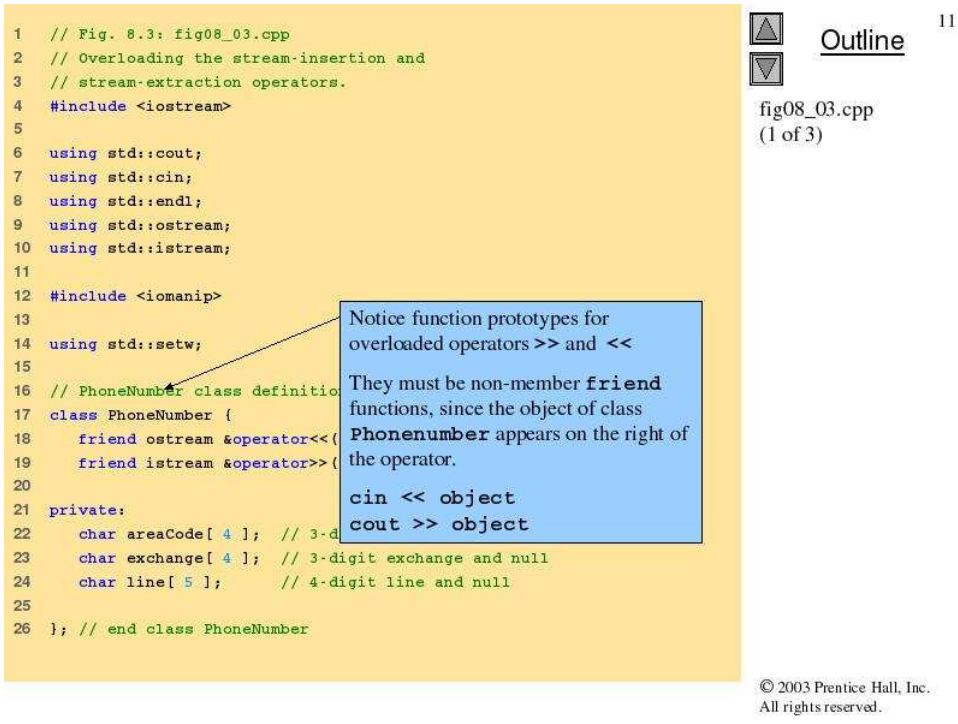
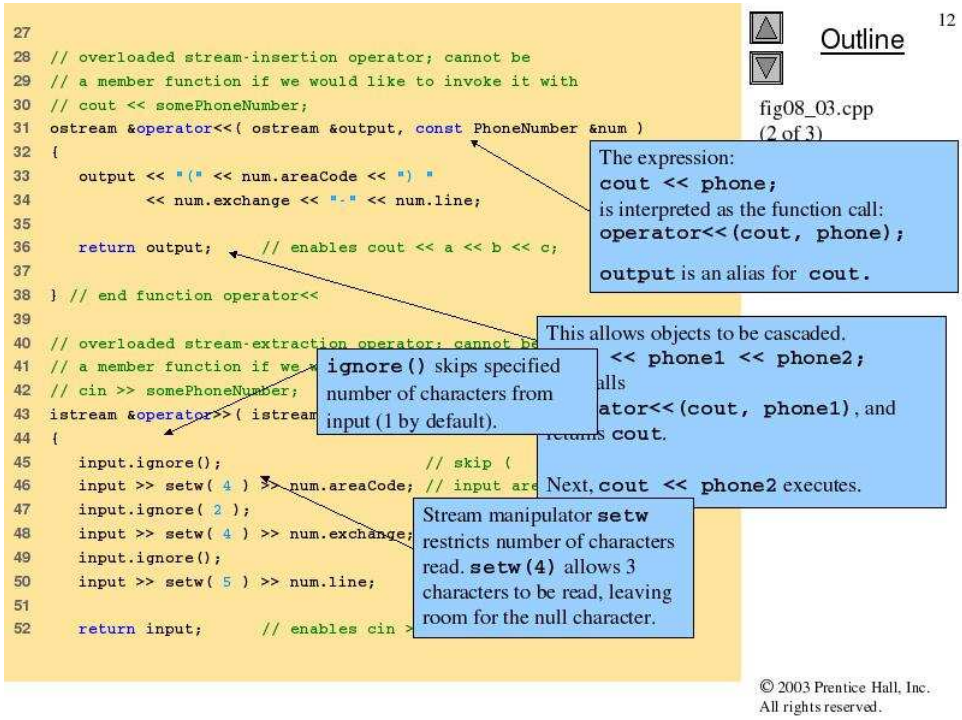


Figure 1: Overloaded stream-insertion and stream extraction operators. (part 1 of 2)



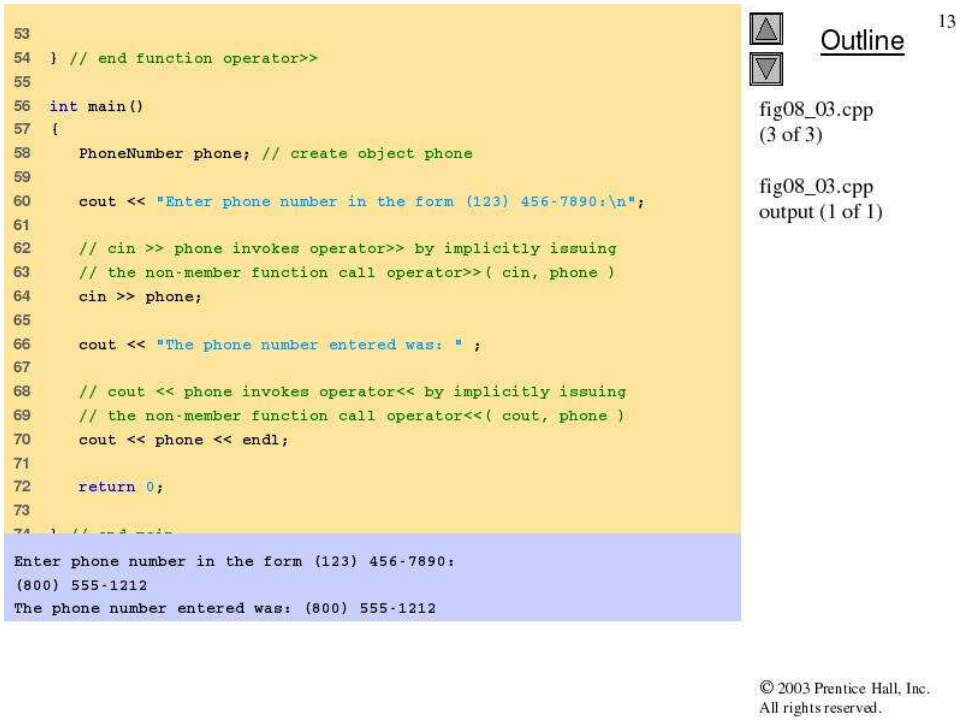


Figure 2: Overloaded stream-insertion and stream extraction operators. (part 2 of 2)

1.6 Overloading Unary Operators

- Overloading unary operators
  - Non-**static** member function, no arguments
  - Non-member function, one argument; Argument must be class object or reference to class object
  - Remember, **static** functions only access **static** data
- Upcoming example (8.10)
  - Overload **!** to test for empty string
  - If non-**static** member function, needs no arguments
    - **!s** becomes **s.operator!()**
    - **class String { public: bool operator!() const; ... };**
- If non-member function, needs one argument
  - **s!** becomes **operator!(s)**
  - **class String { friend bool operator!(const String &) ... }**

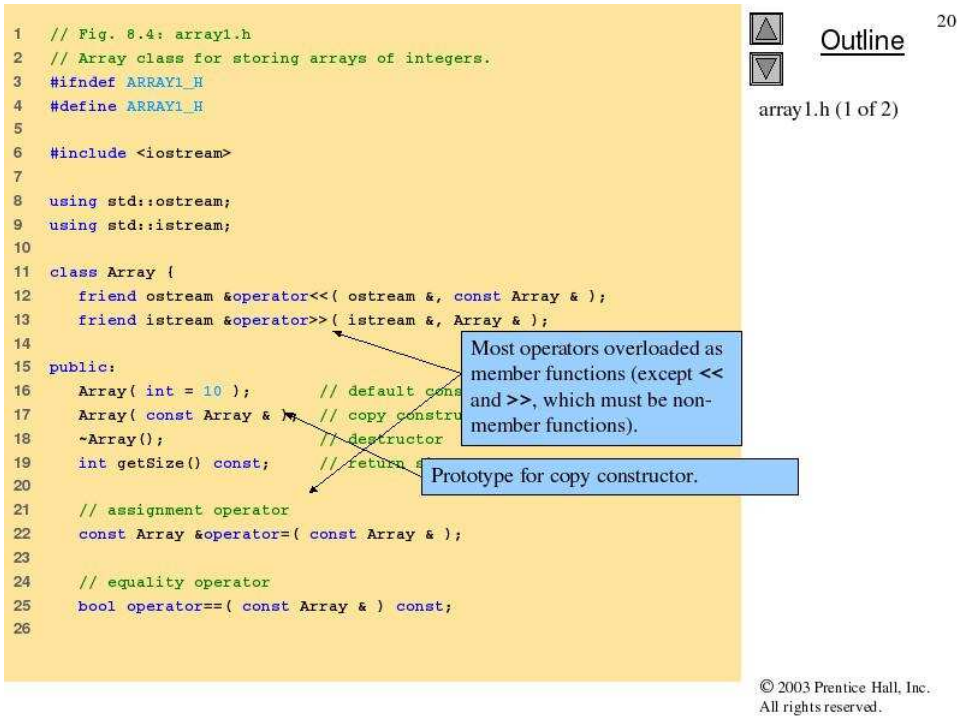
1.7 Overloading Binary Operators

- Overloading binary operators
  - Non-**static** member function, one argument
  - Non-member function, two arguments
  - One argument must be class object or reference
- Upcoming example
  - If non-**static** member function, needs one argument
    - **class String {**
    - **public:**
    - **const String &operator+=(const String &);**
    - **...**
    - **};**
  - **y += z** equivalent to **y.operator+=(z)**

1.8 Case Study: Array class

- Arrays in C++
  - No range checking
  - Cannot be compared meaningfully with ==
  - No array assignment (array names **const** pointers)
  - Cannot input/output entire arrays at once; One element at a time
- Example:Implement an Array class with
  - Range checking
  - Array assignment
  - Arrays that know their size
  - Outputting/inputting entire arrays with << and >>
  - Array comparisons with == and **!=**
- Copy constructor
  - Used whenever copy of object needed
    - Passing by value (return value or parameter)
    - Initializing an object with a copy of another; **Array newArray(oldArray);**
    - **newArray** copy of **oldArray**
  - Prototype for class **Array**
    - **Array(const Array &);**
    - *Must* take reference
      - Otherwise, pass by value
      - Tries to make copy by calling copy constructor ...
      - Infinite loop

The program of Figs. 3-11 demonstrates class **Array** and its overloaded operators.



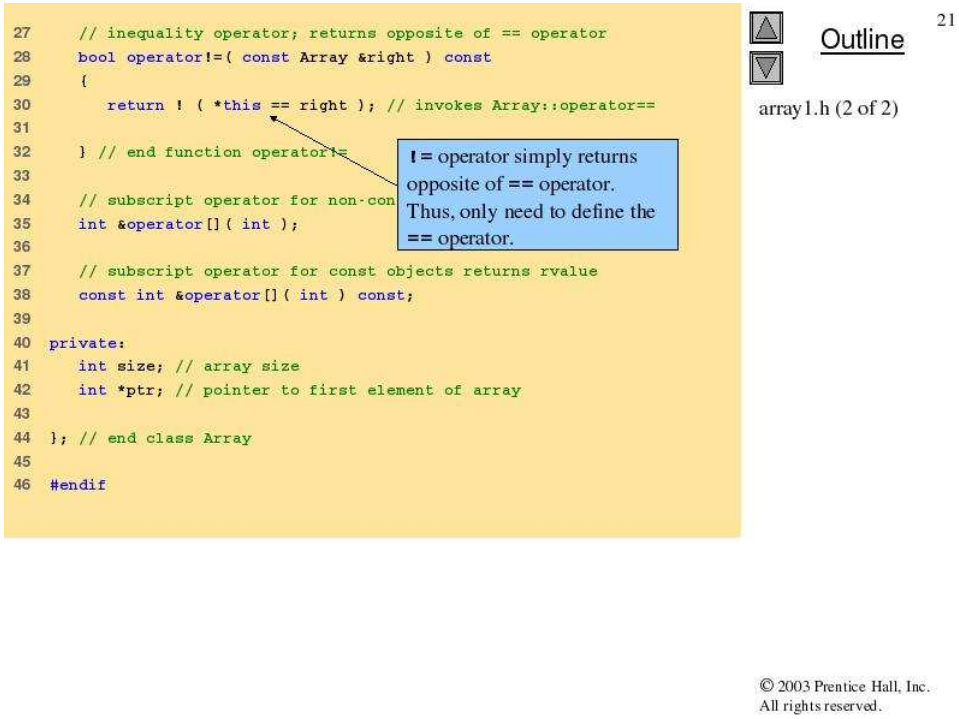


Figure 3: **Array** class definition with overloaded operators.

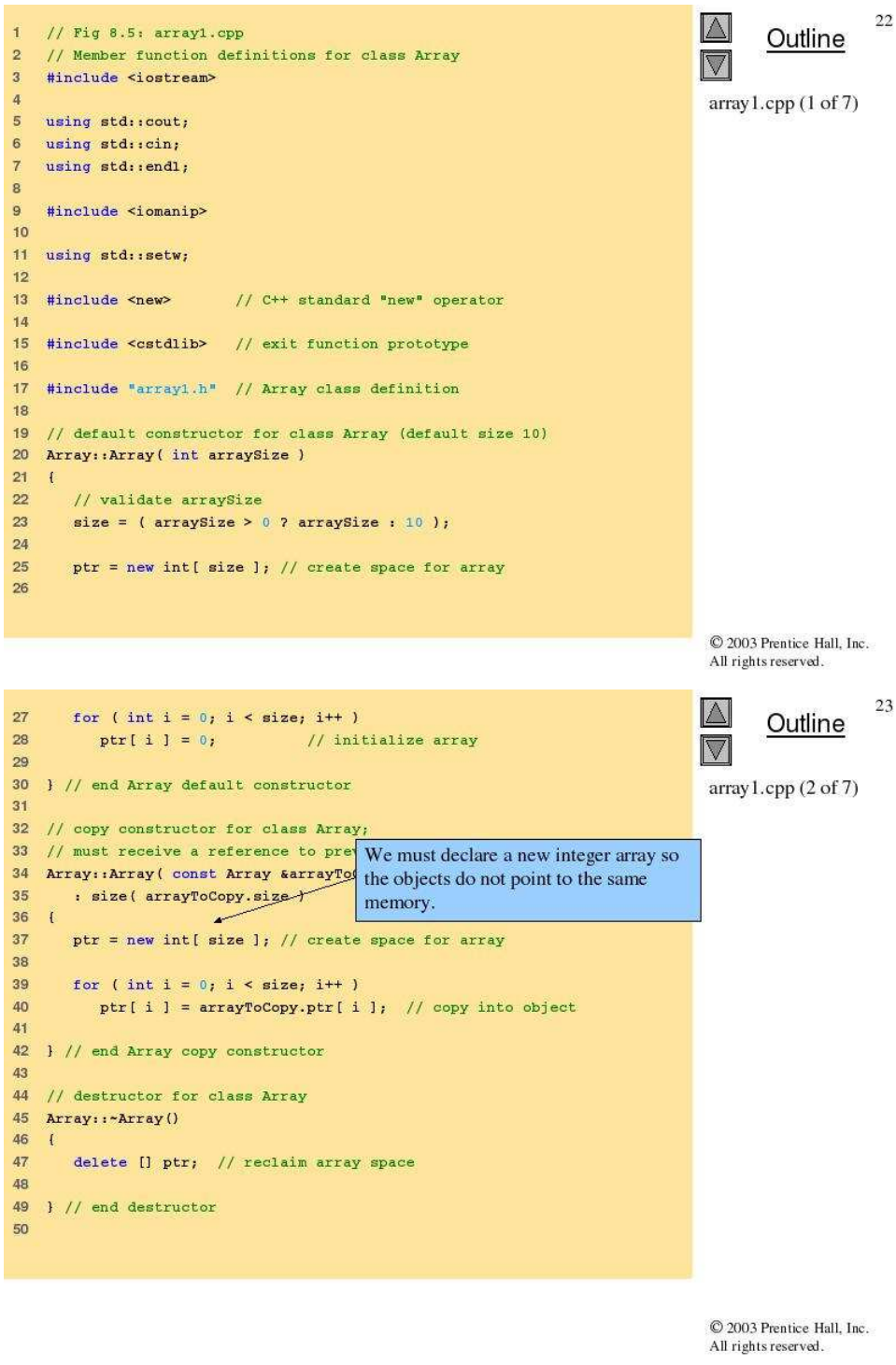


Figure 4: **Array** class member-and friend-function definitions. (part 1 of 4)

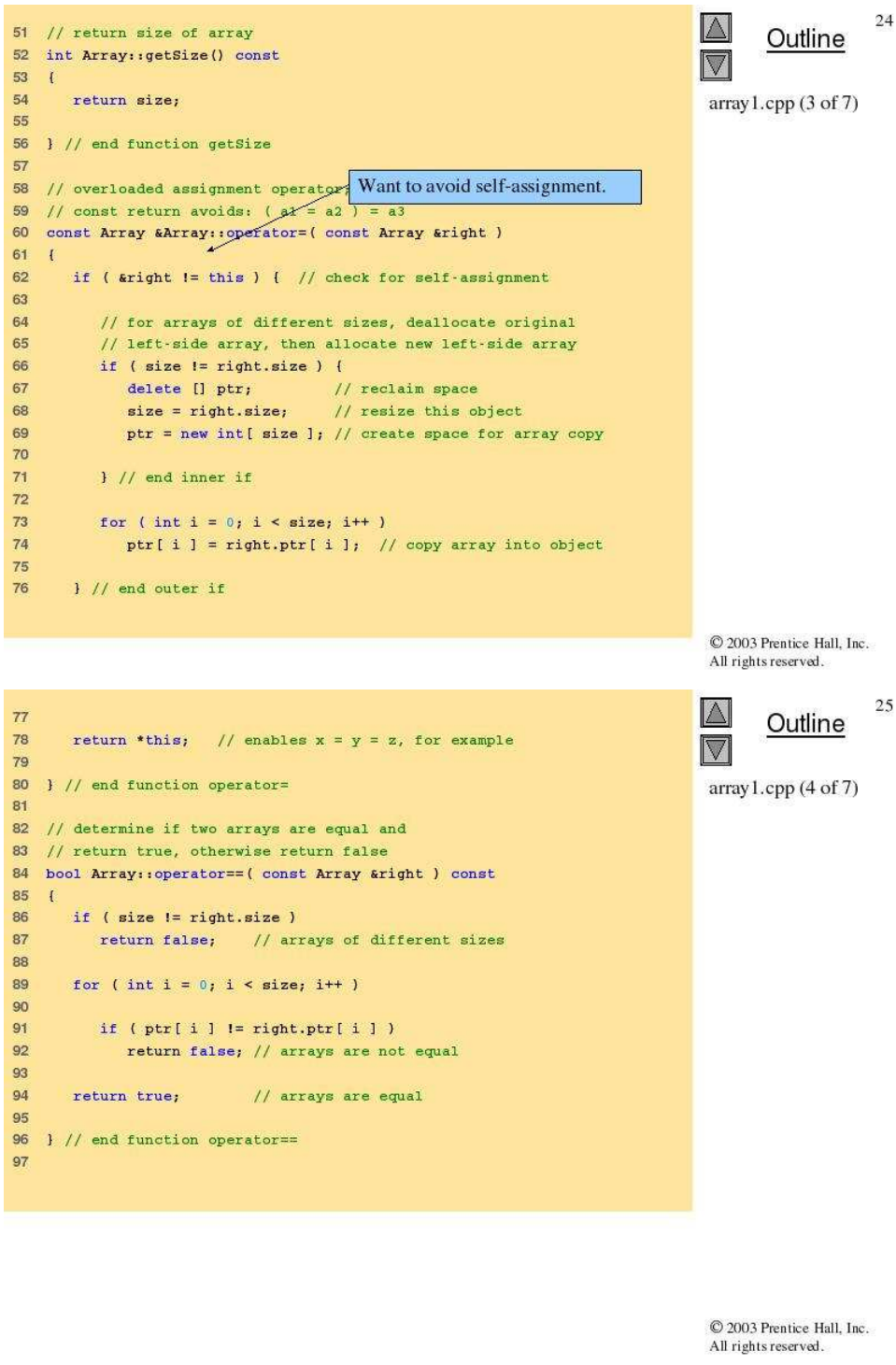
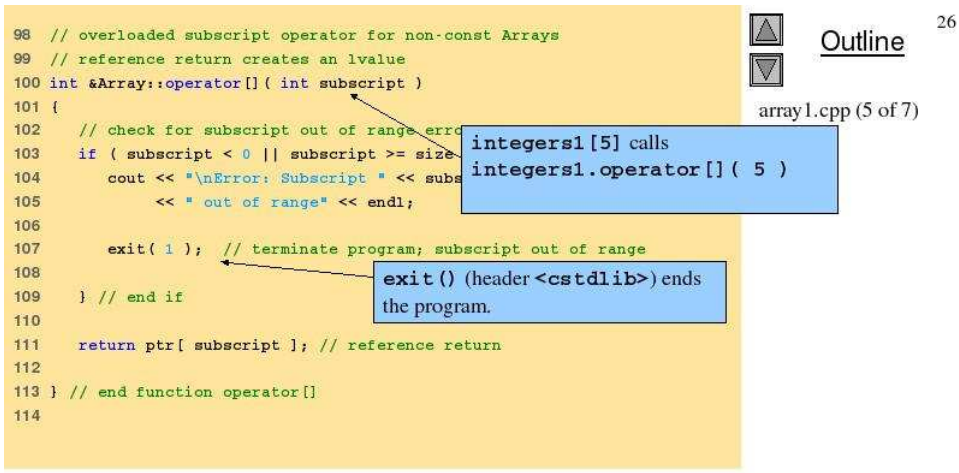


Figure 5: **Array** class member-and friend-function definitions. (part 2 of 4)



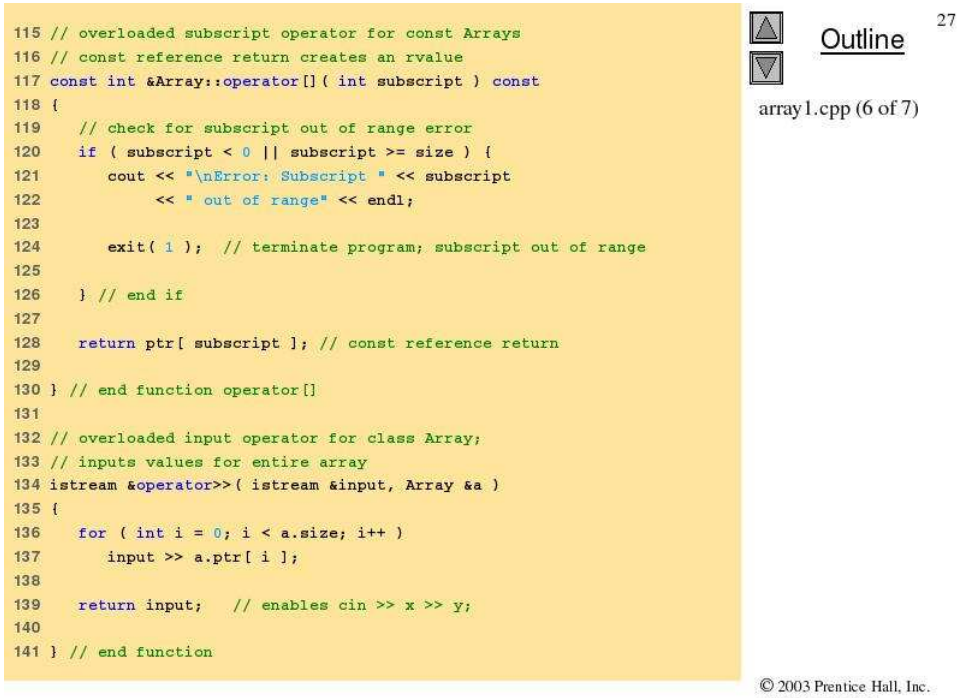
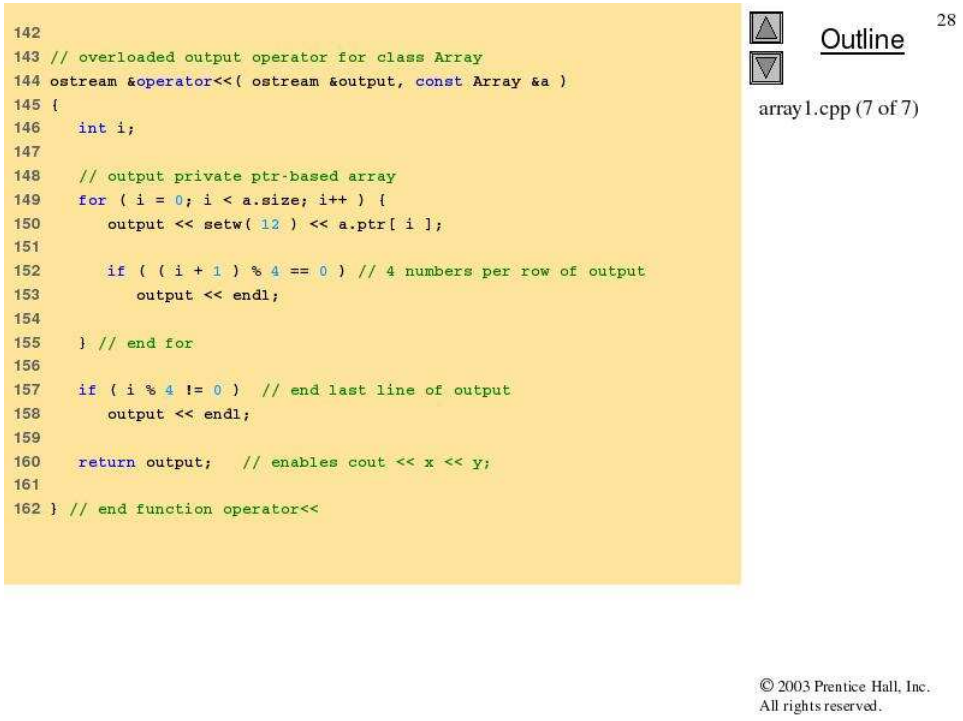


Figure 6: **Array** class member-and friend-function definitions. (part 3 of 4)



```
142
143 // overloaded output operator for class Array
144 ostream &operator<<( ostream &output, const Array &a )
145 {
146    int i;
147
148    // output private ptr-based array
149    for ( i = 0; i < a.size; i++ ) {
150       output << setw( 12 ) << a.ptr[ i ];
151
152       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
153          output << endl;
154
155    } // end for
156
157    if ( i % 4 != 0 )  // end last line of output
158       output << endl;
159
160    return output;   // enables cout << x << y;
161
162 } // end function operator<<
```


Figure 7: Overloaded stream-insertion and stream extraction operators. (part 4 of 2)

Outline

fig08_06.cpp
(1 of 3)

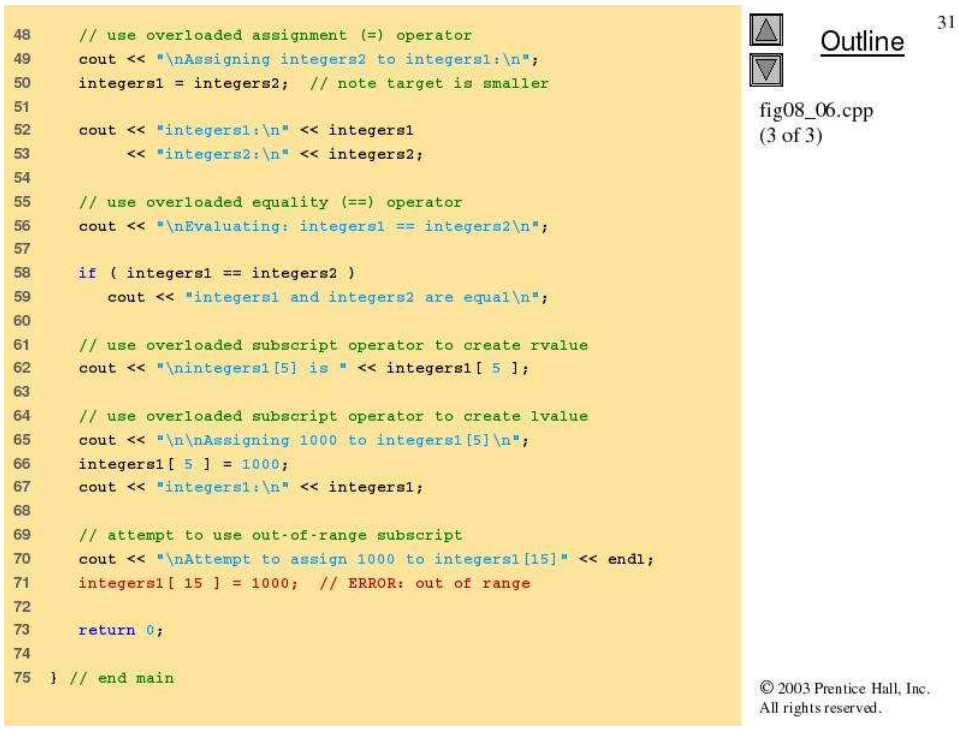
```
1  // Fig. 8.6: fig08_06.cpp
2  // Array class test program.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include "array1.h"
10
11 int main()
12 {
13    Array integers1( 7 );  // seven-element Array
14    Array integers2;       // 10-element Array by default
15
16    // print integers1 size and contents
17    cout << "Size of array integers1 is "
18         << integers1.getSize()
19         << "\nArray after initialization:\n" << integers1;
20
21    // print integers2 size and contents
22    cout << "\nSize of array integers2 is "
23         << integers2.getSize()
24         << "\nArray after initialization:\n" << integers2;
25
```


Outline

fig08_06.cpp
(2 of 3)

```
26    // input and print integers1 and integers2
27    cout << "\nInput 17 integers:\n";
28    cin >> integers1 >> integers2;
29
30    cout << "\nAfter input, the arrays contain:\n"
31         << "integers1:\n" << integers1
32         << "integers2:\n" << integers2;
33
34    // use overloaded inequality (!=) operator
35    cout << "\nEvaluating: integers1 != integers2\n";
36
37    if ( integers1 != integers2 )
38       cout << "integers1 and integers2 are not equal\n";
39
40    // create array integers3 using integers1 as an
41    // initializer; print size and contents
42    Array integers3( integers1 );  // calls copy constructor
43
44    cout << "\nSize of array integers3 is "
45         << integers3.getSize()
46         << "\nArray after initialization:\n" << integers3;
47
```


Figure 8: **Array** class test program. (part 1 of 2)



```
48    // use overloaded assignment (=) operator
49    cout << "\nAssigning integers2 to integers1:\n";
50    integers1 = integers2;  // note target is smaller
51
52    cout << "integers1:\n" << integers1
53         << "integers2:\n" << integers2;
54
55    // use overloaded equality (==) operator
56    cout << "\nEvaluating: integers1 == integers2\n";
57
58    if ( integers1 == integers2 )
59       cout << "integers1 and integers2 are equal\n";
60
61    // use overloaded subscript operator to create rvalue
62    cout << "\nintegers1[5] is " << integers1[ 5 ];
63
64    // use overloaded subscript operator to create lvalue
65    cout << "\n\nAssigning 1000 to integers1[5]\n";
66    integers1[ 5 ] = 1000;
67    cout << "integers1:\n" << integers1;
68
69    // attempt to use out-of-range subscript
70    cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
71    integers1[ 15 ] = 1000;  // ERROR: out of range
72
73    return 0;
74
75 } // end main
```


Figure 9: **Array** class test program. (part 2 of 2)

```
Size of array integers1 is 7
Array after initialization:
           0           0           0           0
           0           0           0

Size of array integers2 is 10
Array after initialization:
           0           0           0           0
           0           0           0           0
           0           0

Input 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the arrays contain:
integers1:
           1           2           3           4
           5           6           7
integers2:
           8           9          10          11
          12          13          14          15
```

Outline

32

fig08_06.cpp
output (1 of 3)


```
Evaluating: integers1 != integers2
integers1 and integers2 are not equal

Size of array integers3 is 7
Array after initialization:
           1           2           3           4
           5           6           7

Assigning integers2 to integers1:
integers1:
           8           9          10          11
          12          13          14          15
          16          17
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13
```

Outline

33

fig08_06.cpp
output (2 of 3)

Figure 10: **Array** class test program, output. (part 1 of 2)

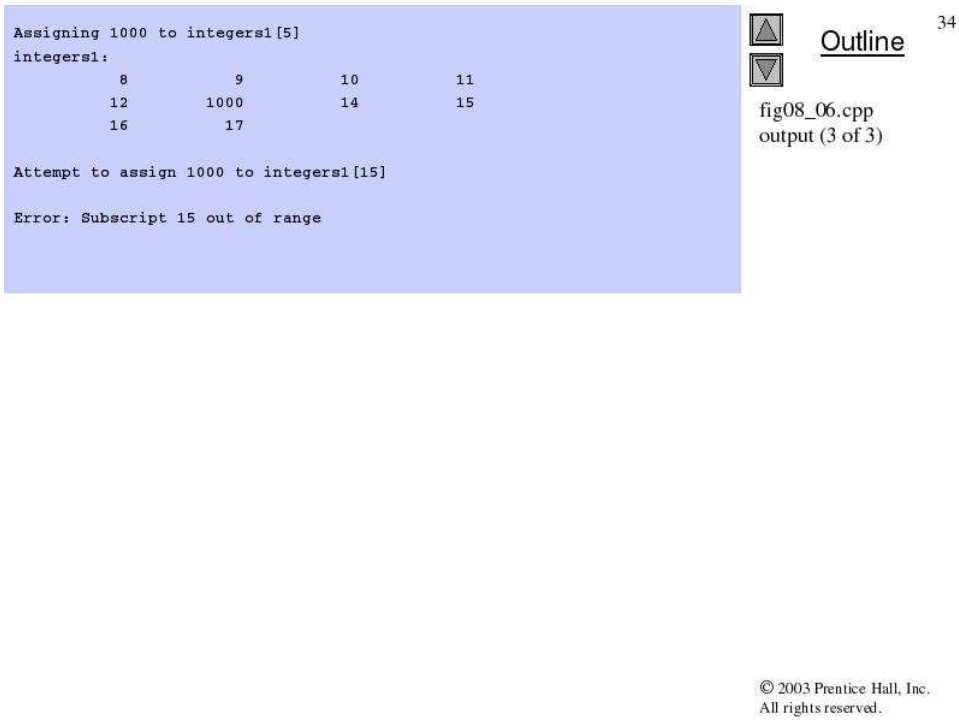


Figure 11: **Array** class test program, output. (part 2 of 2)