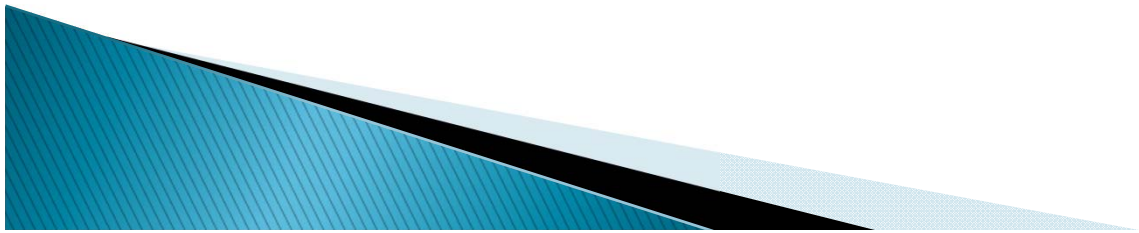
Operator Overloading

C++ How to Program, 9/e

Sources

- C++ How to Program, 9/e (by Paul Deitel and Harvey Deitel)
  - Chapter 10:
    Operator Overloading; Class string



Outline

- Introduction (Section 10.1)
- Case Study: Array Class (Section 10.10)
  - Dynamic Memory Management (Section 10.9)
  - Overloading Binary Operators (Section 10.4)
  - Overloading Unary Operators (Section 10.6)
  - Overloading the Binary Stream Insertion (<<) and Stream Extraction (>>) Operators (Section 10.5)
- Fundamentals of Operator Overloading (Section 10.3)
  - Self-reading
- Converting Between Types (Section 10.12)
- explicit Constructors and Conversion Operators (Section 10.13)

Outline

- **Introduction (Section 10.1)**
- Case Study: Array Class (Section 10.10)
  - Dynamic Memory Management (Section 10.9)
  - Overloading Binary Operators (Section 10.4)
  - Overloading Unary Operators (Section 10.6)
  - Overloading the Binary Stream Insertion (<<) and Stream Extraction (>>) Operators (Section 10.5)
- Fundamentals of Operator Overloading (Section 10.3)
  - Self-reading
- Converting Between Types (Section 10.12)
- explicit Constructors and Conversion Operators (Section 10.13)

Introduction

- This lecture shows how to enable C++'s operators to work with objects
  - a process called operator overloading.
- C++ overloads the addition operator (+) and the subtraction operator (-) to perform differently,
  - depending on their context in integer, floating-point and pointer arithmetic with data of fundamental types.
- You can overload most operators to be used with class objects
  - the compiler generates the appropriate code based on the types of the operands.

Outline

- Introduction (Section 10.1)
- **Case Study: Array Class (Section 10.10)**
  - Dynamic Memory Management (Section 10.9)
  - Overloading Binary Operators (Section 10.4)
  - Overloading Unary Operators (Section 10.6)
  - Overloading the Binary Stream Insertion (<<) and Stream Extraction (>>) Operators (Section 10.5)
- Fundamentals of Operator Overloading (Section 10.3)
  - Self-reading
- Converting Between Types (Section 10.12)
- explicit Constructors and Conversion Operators (Section 10.13)

Pointer-based Arrays: Problems

- A program can easily "walk off" either end of a built-in array,
  - because *C/C++ does not check whether subscripts fall outside the range of the array*.
- When an array is passed to a function, the array's size must be passed as an additional argument.
- Two built-in arrays cannot be meaningfully compared with equality or relational operators.
- One built-in array cannot be assigned to another with the assignment operator.

Case Study: `Array` Class

- With C++, you can implement more robust array capabilities via classes and operator overloading.
- In this section, we'll develop our own custom array class that's preferable to built-in arrays.
- In this example, we create a powerful `Array` class:
  - Performs range checking.
  - Allows one `Array` object to be assigned to another with the assignment operator.
  - Objects know their own size.
  - Input or output entire arrays with the stream extraction (`>>`) and stream insertion (`<<`) operators, respectively.
  - Can compare `Array`s with the equality operators `==` and `!=`.

Input and Output in C++

```
int a;
double b;
char c;
char d[20];

cin >> a >> b >> c >> d;
cout << a << endl
     << b << endl
     << c << endl
     << d << endl;
```

Input and Output in C++ (cont.)

- cout (↔ the printf() function in C):
  - the standard output stream object which is normally "connected" to the screen
  - (<<): the stream insertion operator
- cin (↔ the scanf() function in C):
  - the standard input stream object which is normally "connected" to the keyboard
  - (>>): the stream extraction operator

```
1  // Fig. 10.9: fig10_09.cpp
2  // Array class test program.
3  #include <iostream>
4  #include <stdexcept>
5  #include "Array.h"
6  using namespace std;
7
8  int main()
9  {
10    Array integers1( 7 ); // seven-element Array
11    Array integers2; // 10-element Array by default
12
13    // print integers1 size and contents
14    cout << "Size of Array integers1 is "
15       << integers1.getSize()
16       << "\nArray after initialization:\n" << integers1;
17
18    // print integers2 size and contents
19    cout << "\nSize of Array integers2 is "
20       << integers2.getSize()
21       << "\nArray after initialization:\n" << integers2;
22
23    // input and print integers1 and integers2
24    cout << "\nEnter 17 integers:" << endl;
25    cin >> integers1 >> integers2;
```

Fig. 10.9 | Array class test program. (Part 1 of 7.)

```
26
27       cout << "\nAfter input, the Arrays contain:\n"
28          << "integers1:\n" << integers1
29          << "integers2:\n" << integers2;
30
31       // use overloaded inequality (!=) operator
32       cout << "\nEvaluating: integers1 != integers2" << endl;
33
34       if ( integers1 != integers2 )
35          cout << "integers1 and integers2 are not equal" << endl;
36
37       // create Array integers3 using integers1 as an
38       // initializer; print size and contents
39       Array integers3( integers1 ); // invokes copy constructor
40
41       cout << "\nSize of Array integers3 is "
42          << integers3.getSize()
43          << "\nArray after initialization:\n" << integers3;
44
45       // use overloaded assignment (=) operator
46       cout << "\nAssigning integers2 to integers1:" << endl;
47       integers1 = integers2; // note target Array is smaller
48
```

Fig. 10.9 | Array class test program. (Part 2 of 7.)

```
49       cout << "integers1:\n" << integers1
50          << "integers2:\n" << integers2;
51
52       // use overloaded equality (==) operator
53       cout << "\nEvaluating: integers1 == integers2" << endl;
54
55       if ( integers1 == integers2 )
56          cout << "integers1 and integers2 are equal" << endl;
57
58       // use overloaded subscript operator to create rvalue
59       cout << "\nintegers1[5] is " << integers1[ 5 ];
60
61       // use overloaded subscript operator to create lvalue
62       cout << "\n\nAssigning 1000 to integers1[5]" << endl;
63       integers1[ 5 ] = 1000;
64       cout << "integers1:\n" << integers1;
65
```

Fig. 10.9 | Array class test program. (Part 3 of 7.)

```
66      // attempt to use out-of-range subscript
67      try
68      {
69         cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
70         integers1[ 15 ] = 1000; // ERROR: subscript out of range
71      } // end try
72      catch ( out_of_range &ex )
73      {
74         cout << "An exception occurred: " << ex.what() << endl;
75      } // end catch
76   } // end main
```

Fig. 10.9 | Array class test program. (Part 4 of 7.)

```
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

Enter 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
         16         17
```

Fig. 10.9 | Array class test program. (Part 5 of 7.)

```
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
```

Fig. 10.9 | Array class test program. (Part 6 of 7.)

```
Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13

Assigning 1000 to integers1[5]
integers1:
           8           9          10          11
          12        1000          14          15
          16          17

Attempt to assign 1000 to integers1[15]
An exception occurred: Subscript out of range
```

Fig. 10.9 | Array class test program. (Part 7 of 7.)

```
1  // Fig. 10.10: Array.h
2  // Array class definition with overloaded operators.
3  #ifndef ARRAY_H
4  #define ARRAY_H
5
6  #include <iostream>
7
8  class Array
9  {
10    friend std::ostream &operator<<( std::ostream &, const Array & );
11    friend std::istream &operator>>( std::istream &, Array & );
12
13 public:
14    explicit Array( int = 10 ); // default constructor
15    Array( const Array & ); // copy constructor
16    ~Array(); // destructor
17    size_t getSize() const; // return size
18
19    const Array &operator=( const Array & ); // assignment operator
20    bool operator==( const Array & ) const; // equality operator
21
```

Fig. 10.10 | Array class definition with overloaded operators. (Part 1 of 2.)

```
22      // inequality operator; returns opposite of == operator
23      bool operator!=( const Array &right ) const
24      {
25         return ! ( *this == right ); // invokes Array::operator==
26      } // end function operator!=
27
28      // subscript operator for non-const objects returns modifiable lvalue
29      int &operator[]( int );
30
31      // subscript operator for const objects returns rvalue
32      int operator[]( int ) const;
33   private:
34      size_t size; // pointer-based array size
35      int *ptr; // pointer to first element of pointer-based array
36   }; // end class Array
37
38   #endif
```

Fig. 10.10 | Array class definition with overloaded operators. (Part 2 of 2.)

```
1  // Fig. 10.11: Array.cpp
2  // Array class member- and friend-function definitions.
3  #include <iostream>
4  #include <iomanip>
5  #include <stdexcept>
6
7  #include "Array.h" // Array class definition
8  using namespace std;
9
10 // default constructor for class Array (default size 10)
11 Array::Array( int arraySize )
12    : size( arraySize > 0 ? arraySize :
13          throw invalid_argument( "Array size must be greater than 0" ) ),
14      ptr( new int[ size ] )
15 {
16    for ( size_t i = 0; i < size; ++i )
17       ptr[ i ] = 0; // set pointer-based array element
18 } // end Array default constructor
19
20 // copy constructor for class Array;
21 // must receive a reference to an Array
22 Array::Array( const Array &arrayToCopy )
```

Fig. 10.11 | Array class member- and friend-function definitions. (Part 1 of 6.)

```
23      : size( arrayToCopy.size ),
24        ptr( new int[ size ] )
25   {
26      for ( size_t i = 0; i < size; ++i )
27         ptr[ i ] = arrayToCopy.ptr[ i ]; // copy into object
28   } // end Array copy constructor
29
30   // destructor for class Array
31   Array::~Array()
32   {
33      delete [] ptr; // release pointer-based array space
34   } // end destructor
35
36   // return number of elements of Array
37   size_t Array::getSize() const
38   {
39      return size; // number of elements in Array
40   } // end function getSize
41
```

Fig. 10.11 | Array class member- and friend-function definitions. (Part 2 of 6.)

```
42  // overloaded assignment operator;
43  // const return avoids: ( a1 = a2 ) = a3
44  const Array &Array::operator=( const Array &right )
45  {
46     if ( &right != this ) // avoid self-assignment
47     {
48        // for Arrays of different sizes, deallocate original
49        // left-side Array, then allocate new left-side Array
50        if ( size != right.size )
51        {
52           delete [] ptr; // release space
53           size = right.size; // resize this object
54           ptr = new int[ size ]; // create space for Array copy
55        } // end inner if
56
57        for ( size_t i = 0; i < size; ++i )
58           ptr[ i ] = right.ptr[ i ]; // copy array into object
59     } // end outer if
60
61     return *this; // enables x = y = z, for example
62  } // end function operator=
63
```

Fig. 10.11 | `Array` class member- and `friend`-function definitions. (Part 3 of 6.)

```
64  // determine if two Arrays are equal and
65  // return true, otherwise return false
66  bool Array::operator==( const Array &right ) const
67  {
68     if ( size != right.size )
69        return false; // arrays of different number of elements
70
71     for ( size_t i = 0; i < size; ++i )
72        if ( ptr[ i ] != right.ptr[ i ] )
73           return false; // Array contents are not equal
74
75     return true; // Arrays are equal
76  } // end function operator==
77
78  // overloaded subscript operator for non-const Arrays;
79  // reference return creates a modifiable lvalue
80  int &Array::operator[]( int subscript )
81  {
82     // check for subscript out-of-range error
83     if ( subscript < 0 || subscript >= size )
84        throw out_of_range( "Subscript out of range" );
85
```

Fig. 10.11 | `Array` class member- and `friend`-function definitions. (Part 4 of 6.)

```
86       return ptr[ subscript ]; // reference return
87    } // end function operator[]
88
89    // overloaded subscript operator for const Arrays
90    // const reference return creates an rvalue
91    int Array::operator[]( int subscript ) const
92    {
93       // check for subscript out-of-range error
94       if ( subscript < 0 || subscript >= size )
95          throw out_of_range( "Subscript out of range" );
96
97       return ptr[ subscript ]; // returns copy of this element
98    } // end function operator[]
99
100   // overloaded input operator for class Array;
101   // inputs values for entire Array
102   istream &operator>>( istream &input, Array &a )
103   {
104      for ( size_t i = 0; i < a.size; ++i )
105         input >> a.ptr[ i ];
106
107      return input; // enables cin >> x >> y;
108   } // end function
```

Fig. 10.11 | `Array` class member- and `friend`-function definitions. (Part 5 of 6.)

```
109
110 // overloaded output operator for class Array
111 ostream &operator<<( ostream &output, const Array &a )
112 {
113    // output private ptr-based array
114    for ( size_t i = 0; i < a.size; ++i )
115    {
116       output << setw( 12 ) << a.ptr[ i ];
117
118       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
119          output << endl;
120    } // end for
121
122    if ( a.size % 4 != 0 ) // end last line of output
123       output << endl;
124
125    return output; // enables cout << x << y;
126 } // end function operator<<
```

Fig. 10.11 | `Array` class member- and `friend`-function definitions. (Part 6 of 6.)

Array Default Constructor

- Line 14 of Fig. 10.10 declares the *default constructor* for the class and specifies a default size of 10 elements.
- The default constructor (defined in Fig. 10.11, lines 11–18)
  - validates and assigns the argument to data member `size`,
  - uses `new` to obtain the memory for the internal pointer-based representation of this `Array` and assigns the pointer returned by `new` to data member `ptr`.
  - uses a `for` statement to set all the elements of the array to zero.

```
1  // Fig. 10.11: Array.cpp
2  // Array class member- and friend-function definitions.
3  #include <iostream>
4  #include <iomanip>
5  #include <stdexcept>
6
7  #include "Array.h" // Array class definition
8  using namespace std;
9
10 // default constructor for class Array (default size 10)
11 Array::Array( int arraySize )
12    : size( arraySize > 0 ? arraySize :
13          throw invalid_argument( "Array size must be greater than 0" ) ),
14      ptr( new int[ size ] )
15 {
16    for ( size_t i = 0; i < size; ++i )
17       ptr[ i ] = 0; // set pointer-based array element
18 } // end Array default constructor
19
20 // copy constructor for class Array;
21 // must receive a reference to an Array
22 Array::Array( const Array &arrayToCopy )
```

Fig. 10.11 | Array class member- and friend-function definitions. (Part 1 of 6.)

Dynamic Memory Management

- You can dynamically control the *allocation* and *deallocation* of memory in a program at execution time for data of any built-in or user-defined type.
  - In C, performed with functions:
    - void* malloc (size_t size)
    - void free (void* ptr)
  - In C++, performed with operators:
    - new
    - delete
- The data is created in the free store (also called the heap)
  - a region of memory assigned to each program for storing dynamically allocated data.

Obtaining Dynamic Memory with new

- The `new` operator
  - allocates storage of the proper size for an object,
  - calls the default constructor to initialize the object
  - returns a pointer to the type specified to the right of the `new` operator.
- You can provide an initializer for a newly created fundamental-type variable or object, as in
  - `double *ptr = new double( 3.14159 );`

Releasing Dynamic Memory with delete

- To destroy a dynamically allocated object, use the `delete` operator as follows:
  - `delete ptr;`
- This statement
  - first calls the destructor for the object to which `ptr` points,
  - then deallocates the memory associated with the object, returning the memory to the free store.

Dynamically Allocating Arrays with new []

- You can also use the `new` operator to allocate arrays dynamically.
  - `int *gradesArray = new int[ 10 ]();`
- The parentheses following `new int[10]` initialize the array's elements
  - fundamental numeric types are implicitly set to `0`,
  - `bool`s are implicitly set to `false`,
  - pointers are implicitly set to `nullptr`,
  - class objects are implicitly initialized by their default constructors.

Releasing Dynamically Allocated Arrays with delete []

- To deallocate a dynamically allocated array, use the statement
  - `delete [] ptr;`
- If the pointer points to an array of objects,
  - the statement first calls the destructor for every object in the array,
  - then deallocates the memory.

Array Copy Constructor

- Note line 39 of Fig. 10.9:
  - `Array integers3( integers1 );`
    (or `Array integers3 = integers1;`)
  - This statement invokes the `Array` copy constructor to copy the elements of one `Array` into another.
- Copy constructors are invoked whenever a copy of an object is needed, such as
  - initializing an object with a copy of another object of the same class.
  - in passing an object by value to a function,
  - returning an object by value from a function

```
26
27       cout << "\nAfter input, the Arrays contain:\n"
28          << "integers1:\n" << integers1
29          << "integers2:\n" << integers2;
30
31       // use overloaded inequality (!=) operator
32       cout << "\nEvaluating: integers1 != integers2" << endl;
33
34       if ( integers1 != integers2 )
35          cout << "integers1 and integers2 are not equal" << endl;
36
37       // create Array integers3 using integers1 as an
38       // initializer; print size and contents
39       Array integers3( integers1 ); // invokes copy constructor
40
41       cout << "\nSize of Array integers3 is "
42          << integers3.getSize()
43          << "\nArray after initialization:\n" << integers3;
44
45       // use overloaded assignment (=) operator
46       cout << "\nAssigning integers2 to integers1:" << endl;
47       integers1 = integers2; // note target Array is smaller
48
```

Fig. 10.9 | Array class test program. (Part 2 of 7.)

Array Copy Constructor (cont.)

- The equal sign in the preceding statement is *not* the assignment operator.
  - When an equal sign appears in the declaration of an object, it invokes a constructor for that object.
- Line 15 of Fig. 10.10 declares a *copy constructor* (defined in Fig. 10.11, lines 22–28) that initializes an `Array` by making a copy of an existing `Array` object.

```
1  // Fig. 10.11: Array.cpp
2  // Array class member- and friend-function definitions.
3  #include <iostream>
4  #include <iomanip>
5  #include <stdexcept>
6
7  #include "Array.h" // Array class definition
8  using namespace std;
9
10 // default constructor for class Array (default size 10)
11 Array::Array( int arraySize )
12    : size( arraySize > 0 ? arraySize :
13          throw invalid_argument( "Array size must be greater than 0" ) ),
14      ptr( new int[ size ] )
15 {
16    for ( size_t i = 0; i < size; ++i )
17       ptr[ i ] = 0; // set pointer-based array element
18 } // end Array default constructor
19
20 // copy constructor for class Array;
21 // must receive a reference to an Array
22 Array::Array( const Array &arrayToCopy )
```

Fig. 10.11 | Array class member- and friend-function definitions. (Part 1 of 6.)

```
23      : size( arrayToCopy.size ),
24        ptr( new int[ size ] )
25   {
26      for ( size_t i = 0; i < size; ++i )
27         ptr[ i ] = arrayToCopy.ptr[ i ]; // copy into object
28   } // end Array copy constructor
29
30   // destructor for class Array
31   Array::~Array()
32   {
33      delete [] ptr; // release pointer-based array space
34   } // end destructor
35
36   // return number of elements of Array
37   size_t Array::getSize() const
38   {
39      return size; // number of elements in Array
40   } // end function getSize
41
```

Fig. 10.11 | Array class member- and friend-function definitions. (Part 2 of 6.)

Array Copy Constructor (cont.)

- In fact, C++ automatically provides a copy constructor for you if you do not explicitly define one.
  - simply perform the member-wise copying
  - the problem of leaving both `Array` objects pointing to the same dynamically allocated memory
- Implicit member functions generated by the compiler:
  - default constructor if you define no constructors
  - copy constructor if you don't define one
  - assignment operator if you don't define one
  - default destructor if you don't define one
  - address operator if you don't define one

Array Destructor

- Line 16 of Fig. 10.10 declares the class's destructor (defined in Fig. 10.11, lines 31–34).
- The destructor is invoked when an object of class `Array` goes out of scope.
- The destructor uses `delete []` to release the memory allocated dynamically by `new` in the constructor.

```
23      : size( arrayToCopy.size ),
24        ptr( new int[ size ] )
25   {
26      for ( size_t i = 0; i < size; ++i )
27         ptr[ i ] = arrayToCopy.ptr[ i ]; // copy into object
28   } // end Array copy constructor
29
30   // destructor for class Array
31   Array::~Array()
32   {
33      delete [] ptr; // release pointer-based array space
34   } // end destructor
35
36   // return number of elements of Array
37   size_t Array::getSize() const
38   {
39      return size; // number of elements in Array
40   } // end function getSize
41
```

Fig. 10.11 | Array class member- and friend-function definitions. (Part 2 of 6.)

Overloading Binary Operators

- A binary operator can be overloaded
  - as a member function with one parameter, or
  - as a non-member function with two parameters (one of those parameters must be either an object of the class or a reference to an object of the class)

Overloading Binary Operators (cont.)

- Overloading binary operator < as a member function

```
class XXX
{
public:
     bool operator<( const XXX & ) const;
     ...
};
```

  - If y and z are XXX-class objects, then
    - `y < z` is treated as if `y.operator<(z)` had been written.

Overloading Binary Operators (cont.)

- Overloading binary operator < as a non-member function
  - Must take two arguments
  - One of which must be an object of the associated class

```
class XXX
{
    ...
};
bool operator<( const XXX &, const XXX & );
```

  - If y and z are XXX-class objects, then
    - y < z is treated as if the call operator<(y, z) had been written in the program.

Overloading Unary Operators

- A unary operator for a class can be overloaded
  - as a member function with no arguments or
  - as a non-member function with one argument that must be an object (or a reference to an object) of the class.

Overloading Unary Operators (cont.)

- Overloading unary operator ! as a member function

```
class YYY
{
public:
     bool operator!() const;
     ...
};
```

  - If `s` is a `YYY`-class object, then
    - `!s` is treated as if `s.operator!()` had been written.

Overloading Unary Operators (cont.)

- Overloading unary operator ! as a non-member function

```
class YYY
{
    ...
};
bool operator!( const YYY & );
```

  - If `s` is a YYY-class object, then
    - `!s` is treated as if the call `operator!(s)` had been written.

Overloaded Operators in Array

- Assignment operator
  - `const Array & Array::operator=( const Array & );`
- Equality operator
  - `bool Array::operator==( const Array & ) const;`
- Inequality operator
  - `bool Array::operator!=( const Array & ) const;`
- Subscript operator
  - `int & Array::operator[]( int );` //non-const objects
  - `int Array::operator[]( int ) const;` //const objects

```
42  // overloaded assignment operator;
43  // const return avoids: ( a1 = a2 ) = a3
44  const Array &Array::operator=( const Array &right )
45  {
46     if ( &right != this ) // avoid self-assignment
47     {
48        // for Arrays of different sizes, deallocate original
49        // left-side Array, then allocate new left-side Array
50        if ( size != right.size )
51        {
52           delete [] ptr; // release space
53           size = right.size; // resize this object
54           ptr = new int[ size ]; // create space for Array copy
55        } // end inner if
56
57        for ( size_t i = 0; i < size; ++i )
58           ptr[ i ] = right.ptr[ i ]; // copy array into object
59     } // end outer if
60
61     return *this; // enables x = y = z, for example
62  } // end function operator=
63
```

Fig. 10.11 | Array class member- and friend-function definitions. (Part 3 of 6.)

```
64  // determine if two Arrays are equal and
65  // return true, otherwise return false
66  bool Array::operator==( const Array &right ) const
67  {
68     if ( size != right.size )
69        return false; // arrays of different number of elements
70
71     for ( size_t i = 0; i < size; ++i )
72        if ( ptr[ i ] != right.ptr[ i ] )
73           return false; // Array contents are not equal
74
75     return true; // Arrays are equal
76  } // end function operator==
77
78  // overloaded subscript operator for non-const Arrays;
79  // reference return creates a modifiable lvalue
80  int &Array::operator[]( int subscript )
81  {
82     // check for subscript out-of-range error
83     if ( subscript < 0 || subscript >= size )
84        throw out_of_range( "Subscript out of range" );
85
```

Fig. 10.11 | `Array` class member- and `friend`-function definitions. (Part 4 of 6.)

```
86       return ptr[ subscript ]; // reference return
87    } // end function operator[]
88
89    // overloaded subscript operator for const Arrays
90    // const reference return creates an rvalue
91    int Array::operator[]( int subscript ) const
92    {
93       // check for subscript out-of-range error
94       if ( subscript < 0 || subscript >= size )
95          throw out_of_range( "Subscript out of range" );
96
97       return ptr[ subscript ]; // returns copy of this element
98    } // end function operator[]
99
100   // overloaded input operator for class Array;
101   // inputs values for entire Array
102   istream &operator>>( istream &input, Array &a )
103   {
104      for ( size_t i = 0; i < a.size; ++i )
105         input >> a.ptr[ i ];
106
107      return input; // enables cin >> x >> y;
108   } // end function
```

Fig. 10.11 | `Array` class member- and `friend`-function definitions. (Part 5 of 6.)

Overloaded Subscript Operators

- Lines 29 and 32 of Fig. 10.10 declare two overloaded subscript operators (defined in Fig. 10.11 in lines 80–87 and 91–98).
- The compiler creates a call to the `const` version of `operator[]` (Fig. 10.11, lines 91–98) when
  - the subscript operator is used on a `const Array` object.

Overloading the Binary Stream Insertion and Stream Extraction Operators

- You can input and output fundamental-type data using the stream extraction operator >> and the stream insertion operator <<.
- The C++ class libraries overload these binary operators for each fundamental type, including pointers and `char *` strings.
- You can also overload these operators to perform input and output for your own types.

Overloaded >> and << Operators as Non-Member friend Functions

- Define the functions
  - `operator>>` and
  - `operator<<`,

  and declare them in `Array` as non-member, `friend` functions.
- They're *non-member functions* because the object of class `Array` is the operator's *right* operand.

```
1  // Fig. 10.10: Array.h
2  // Array class definition with overloaded operators.
3  #ifndef ARRAY_H
4  #define ARRAY_H
5
6  #include <iostream>
7
8  class Array
9  {
10    friend std::ostream &operator<<( std::ostream &, const Array & );
11    friend std::istream &operator>>( std::istream &, Array & );
12
13 public:
14    explicit Array( int = 10 ); // default constructor
15    Array( const Array & ); // copy constructor
16    ~Array(); // destructor
17    size_t getSize() const; // return size
18
19    const Array &operator=( const Array & ); // assignment operator
20    bool operator==( const Array & ) const; // equality operator
21
```

Fig. 10.10 | Array class definition with overloaded operators. (Part 1 of 2.)

Overloaded >> and << Operators as Non-Member friend Functions (cont.)

- When the compiler sees an expression like `cout << arrayObject`, it invokes
  - `operator<<( cout, arrayObject )`
  - `cout` has type `ostream`
- When the compiler sees an expression like `cin >> arrayObject`, it invokes
  - `operator>>( cin, arrayObject )`
  - `cin` has type `istream`

```
86       return ptr[ subscript ]; // reference return
87    } // end function operator[]
88
89    // overloaded subscript operator for const Arrays
90    // const reference return creates an rvalue
91    int Array::operator[]( int subscript ) const
92    {
93       // check for subscript out-of-range error
94       if ( subscript < 0 || subscript >= size )
95          throw out_of_range( "Subscript out of range" );
96
97       return ptr[ subscript ]; // returns copy of this element
98    } // end function operator[]
99
100   // overloaded input operator for class Array;
101   // inputs values for entire Array
102   istream &operator>>( istream &input, Array &a )
103   {
104      for ( size_t i = 0; i < a.size; ++i )
105         input >> a.ptr[ i ];
106
107      return input; // enables cin >> x >> y;
108   } // end function
```

Fig. 10.11 | `Array` class member- and `friend`-function definitions. (Part 5 of 6.)

```
109
110 // overloaded output operator for class Array
111 ostream &operator<<( ostream &output, const Array &a )
112 {
113    // output private ptr-based array
114    for ( size_t i = 0; i < a.size; ++i )
115    {
116       output << setw( 12 ) << a.ptr[ i ];
117
118       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
119          output << endl;
120    } // end for
121
122    if ( a.size % 4 != 0 ) // end last line of output
123       output << endl;
124
125    return output; // enables cout << x << y;
126 } // end function operator<<
```

Fig. 10.11 | `Array` class member- and `friend`-function definitions. (Part 6 of 6.)

Operators as Member vs. Non-Member Functions

- When an operator function is implemented as a member function,
  - the leftmost (or only) operand must be an object (or a reference to an object) of the operator's class.
- If the left operand *must* be an object of a different class or a fundamental type,
  - this operator function *must* be implemented as a non-member function.

Outline

- Introduction (Section 10.1)
- Case Study: Array Class (Section 10.10)
  - Dynamic Memory Management (Section 10.9)
  - Overloading Binary Operators (Section 10.4)
  - Overloading Unary Operators (Section 10.6)
  - Overloading the Binary Stream Insertion (<<) and Stream Extraction (>>) Operators (Section 10.5)
- Fundamentals of Operator Overloading (Section 10.3)
  - Self-reading
- **Converting Between Types (Section 10.12)**
- explicit Constructors and Conversion Operators (Section 10.13)

Converting Between Types

- How to convert among user-defined types, and between user-defined types and fundamental types?
  - The compiler cannot know in advance.
  - You must specify how to do this.

Conversion Constructors

- Conversion constructors:
  - constructors that can be called with a single argument
- Conversion constructors can turn objects of other types (including fundamental types) into objects of a particular class.

Other type -> Conversion Constructor -> The present class

Conversion Constructors (cont.)

- In the `Array` class, the constructor

```
Array( int );
```

  serves as the conversion constructor that supports the following statement

```
Array int_arr = 20;
```

- This is equivalent to the other two forms:
  - `Array int_arr(20);`
  - `Array int_arr = Array(20);`

Conversion Operators or Functions

- A conversion operator (also called a *cast operator* or *conversion function*) can be used to convert an object of one class to another type.

  The present class -> Conversion Operator -> Other type

  - Such a conversion operator must be a *non-`static` member function.*

- For example,

```
MyClass::operator char *() const;
```

  - declares a cast operator function for converting an object of class `MyClass` into a `char *` data object.

Outline

- Introduction (Section 10.1)
- Case Study: Array Class (Section 10.10)
  - Dynamic Memory Management (Section 10.9)
  - Overloading Binary Operators (Section 10.4)
  - Overloading Unary Operators (Section 10.6)
  - Overloading the Binary Stream Insertion (<<) and Stream Extraction (>>) Operators (Section 10.5)
- Fundamentals of Operator Overloading (Section 10.3)
  - Self-reading
- Converting Between Types (Section 10.12)
- **explicit Constructors and Conversion Operators (Section 10.13)**

`explicit` Constructors and Conversion Operators

- Recall that we've been declaring as explicit every constructor that can be called with one argument.
- With the exception of copy constructors, any constructor that can be called with a single argument and is not declared explicit can be used by the compiler to perform an implicit conversion.
  - The conversion is automatic and you need not use a cast operator.
  - In some situations, implicit conversions are undesirable or error-prone.

```
1  // Fig. 10.12: fig10_12.cpp
2  // Single-argument constructors and implicit conversions.
3  #include <iostream>
4  #include "Array.h"
5  using namespace std;
6
7  void outputArray( const Array & ); // prototype
8
9  int main()
10 {
11    Array integers1( 7 ); // 7-element Array
12    outputArray( integers1 ); // output Array integers1
13    outputArray( 3 ); // convert 3 to an Array and output Array's contents
14 } // end main
15
16 // print Array contents
17 void outputArray( const Array &arrayToOutput )
18 {
19    cout << "The Array received has " << arrayToOutput.getSize()
20       << " elements. The contents are:\n" << arrayToOutput << endl;
21 } // end outputArray
```

Fig. 10.12 | Single-argument constructors and implicit conversions. (Part 1 of 2.)

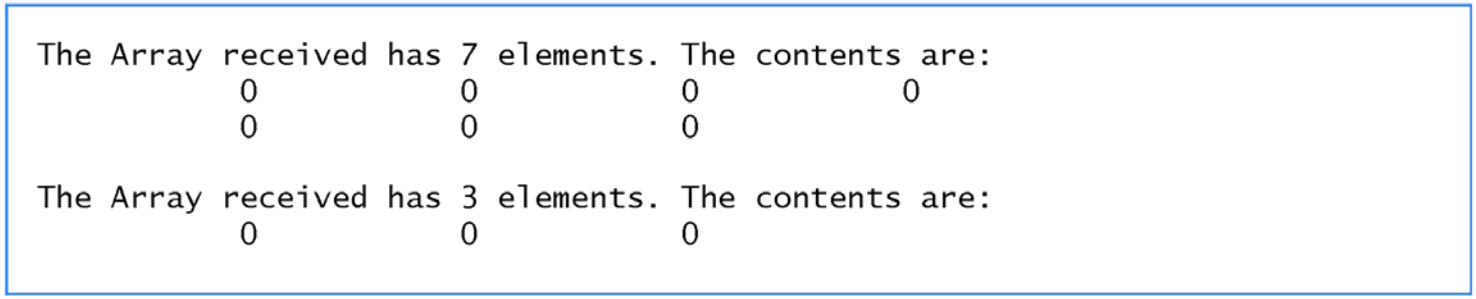


Fig. 10.12 | Single-argument constructors and implicit conversions. (Part 2 of 2.)

Preventing Implicit Conversions with Single-Argument Constructors

- The reason we've been declaring every single-argument constructor preceded by the keyword `explicit` is to
  - suppress implicit conversions via conversion constructors when such conversions should not be allowed.

```
1  // Fig. 10.13: fig10_13.cpp
2  // Demonstrating an explicit constructor.
3  #include <iostream>
4  #include "Array.h"
5  using namespace std;
6
7  void outputArray( const Array & ); // prototype
8
9  int main()
10 {
11    Array integers1( 7 ); // 7-element Array
12    outputArray( integers1 ); // output Array integers1
13    outputArray( 3 ); // convert 3 to an Array and output Array's contents
14    outputArray( Array( 3 ) ); // explicit single-argument constructor call
15 } // end main
16
17 // print Array contents
18 void outputArray( const Array &arrayToOutput )
19 {
20    cout << "The Array received has " << arrayToOutput.getSize()
21       << " elements. The contents are:\n" << arrayToOutput << endl;
22 } // end outputArray
```

Fig. 10.13 | Demonstrating an `explicit` constructor. (Part 1 of 2.)

```
c:\books\2012\cpphtp9\examples\ch10\fig10_13\fig10_13.cpp(13): error C2664:
'outputArray' : cannot convert parameter 1 from 'int' to 'const Array &'
          Reason: cannot convert from 'int' to 'const Array'
          Constructor for class 'Array' is declared 'explicit'
```

Fig. 10.13 | Demonstrating an `explicit` constructor. (Part 2 of 2.)

explicit Conversion Operators

- Similar to declaring single-argument constructors `explicit`, you can
  - declare conversion operators explicit to prevent the compiler from using them to perform implicit conversions.
- For example, the prototype:

```
explicit MyClass::operator char
*() const;
```

declares `MyClass`'s `char *` cast operator `explicit`.