```
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.)

10.10 Case Study: `Array` Class (cont.)

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`.
- Then the constructor uses a `for` statement to set all the elements of the array to zero.

10.10 Case Study: `Array` Class (cont.)

***Array* Copy Constructor**

- 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.
- *Such copying must be done carefully to avoid the pitfall of leaving both `Array` objects pointing to the same dynamically allocated memory.*
- This is exactly the problem that would occur with default memberwise copying, if the compiler is allowed to define a default copy constructor for this class.
- Copy constructors are invoked whenever a copy of an object is needed, such as in passing an object by value to a function, returning an object by value from a function or initializing an object with a copy of another object of the same class.

10.10 Case Study: `Array` Class (cont.)

- The copy constructor for `Array` copies the `size` of the initializer `Array` into 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`.
- Then the copy constructor uses a `for` statement to copy all the elements of the initializer `Array` into the new `Array` object.
- An object of a class can look at the `private` data of any other object of that class (using a handle that indicates which object to access).

Software Engineering Observation 10.3

The argument to a copy constructor should be a `const` reference to allow a `const` object to be copied.

Common Programming Error 10.4

If the copy constructor simply copied the pointer in the source object to the target object's pointer, then both would point to the same dynamically allocated memory. The first destructor to execute would delete the dynamically allocated memory, and the other object's `ptr` would point to memory that's no longer allocated, a situation called a dangling pointer—this would likely result in a serious runtime error (such as early program termination) when the pointer was used.

10.10 Case Study: `Array` Class (cont.)

***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.

Error-Prevention Tip 10.3

If after deleting dynamically allocated memory, the pointer will continue to exist in memory, set the pointer's value to `nullptr` to indicate that the pointer no longer points to memory in the free store. By setting the pointer to `nullptr`, the program loses access to that free-store space, which could be reallocated for a different purpose. If you do not set the pointer to `nullptr`, your code could inadvertently access the reallocated memory, causing subtle, nonrepeatable logic errors. We did not set `ptr` to `nullptr` in line 33 of Fig. 10.11 because after the destructor executes, the `Array` object no longer exists in memory.

10.10 Case Study: `Array` Class (cont.)

Overloaded Assignment Operator

- Line 19 of Fig. 10.10 declares the overloaded assignment operator function for the class.
- When the compiler sees the expression `integers1 = integers2` in line 47 of Fig. 10.9, the compiler invokes member function `operator=` with the call
  - `integers1.operator=( integers2 )`
- Member function `operator=`'s implementation (Fig. 10.11, lines 44–62) tests for self-assignment (line 46) in which an `Array` object is being assigned to itself.
- When `this` is equal to the `right` operand's address, a *self-assignment* is being attempted, so the assignment is skipped.

10.10 Case Study: `Array` Class (cont.)

- `operator=` determines whether the sizes of the two `Array`s are identical (line 50); in that case, the original array of integers in the left-side `Array` object is *not* reallocated.
- Otherwise, `operator=` uses `delete []` (line 52) to release the memory, copies the `size` of the source array to the `size` of the target `Array` (line 53), uses `new` to allocate memory for the target `Array` and places the pointer returned by `new` into the `Array`'s `ptr` member.
- Regardless of whether this is a self-assignment, the member function returns the current object (i.e., `*this` in line 61) as a constant reference; this enables cascaded `Array` assignments such as `x = y = z`, but prevents ones like `(x = y) = z` because `z` cannot be assigned to the `const Array&` reference that is returned by `(x = y)`.

Software Engineering Observation 10.4

A copy constructor, a destructor and an overloaded assignment operator are usually provided as a group for any class that uses dynamically allocated memory. With the addition of move semantics in C++11, other functions should also be provided, as you'll see in Chapter 24.