Instructor Note: *C How to Program*, Chapter 18 is a copy of *C++ How to Program, 9/e* Chapter 10. We have not renumbered the PowerPoint Slides.

Chapter 10
Operator Overloading; Class `string`

C++ How to Program, 9/e, GE

OBJECTIVES

In this chapter you'll:

- Learn how operator overloading can help you craft valuable classes.
- Overload unary and binary operators.
- Convert objects from one class to another class.
- Use overloaded operators and additional features of the `string` class.
- Create `PhoneNumber`, `Date` and `Array` classes that provide overloaded operators.
- Perform dynamic memory allocation with `new` and `delete`.
- Use keyword `explicit` to indicate that a constructor cannot be used for implicit conversions.
- Experience a "light-bulb moment" when you'll truly appreciate the elegance and beauty of the class concept.

10.1 Introduction

- This chapter shows how to enable C++'s operators to work with objects—a process called operator overloading.
- One example of an overloaded operator built into C++ is <<, which is used *both* as the stream insertion operator and as the bitwise left-shift operator.
- 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.

10.2 Using the Overloaded Operators of Standard Library Class `string`

- Figure 10.1 demonstrates many of class `string`'s overloaded operators and several other useful member functions, including `empty`, `substr` and `at`.
- Function `empty` determines whether a `string` is empty, function `substr` returns a `string` that represents a portion of an existing `string` and function `at` returns the character at a specific index in a `string` (after checking that the index is in range).
- Chapter 21 presents class string in detail.

```
1  // Fig. 10.1: fig10_01.cpp
2  // Standard Library string class test program.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string s1( "happy" );
10    string s2( " birthday" );
11    string s3;
12
13    // test overloaded equality and relational operators
14    cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
15       << "\"; s3 is \"" << s3 << '\"'
16       << "\n\nThe results of comparing s2 and s1:"
17       << "\ns2 == s1 yields " << ( s2 == s1 ? "true" : "false" )
18       << "\ns2 != s1 yields " << ( s2 != s1 ? "true" : "false" )
19       << "\ns2 >  s1 yields " << ( s2 > s1 ? "true" : "false" )
20       << "\ns2 <  s1 yields " << ( s2 < s1 ? "true" : "false" )
21       << "\ns2 >= s1 yields " << ( s2 >= s1 ? "true" : "false" )
22       << "\ns2 <= s1 yields " << ( s2 <= s1 ? "true" : "false" );
23
```

Fig. 10.1 | Standard Library `string` class test program. (Part 1 of 6.)

```
24      // test string member-function empty
25      cout << "\n\nTesting s3.empty():" << endl;
26
27      if ( s3.empty() )
28      {
29         cout << "s3 is empty; assigning s1 to s3;" << endl;
30         s3 = s1; // assign s1 to s3
31         cout << "s3 is \"" << s3 << "\"";
32      } // end if
33
34      // test overloaded string concatenation operator
35      cout << "\n\ns1 += s2 yields s1 = ";
36      s1 += s2; // test overloaded concatenation
37      cout << s1;
38
39      // test overloaded string concatenation operator with a C string
40      cout << "\n\ns1 += \" to you\" yields" << endl;
41      s1 += " to you";
42      cout << "s1 = " << s1 << "\n\n";
43
44      // test string member function substr
45      cout << "The substring of s1 starting at location 0 for\n"
46         << "14 characters, s1.substr(0, 14), is:\n"
47         << s1.substr( 0, 14 ) << "\n\n";
```

Fig. 10.1 | Standard Library `string` class test program. (Part 2 of 6.)

```
48
49      // test substr "to-end-of-string" option
50      cout << "The substring of s1 starting at\n"
51         << "location 15, s1.substr(15), is:\n"
52         << s1.substr( 15 ) << endl;
53
54      // test copy constructor
55      string s4( s1 );
56      cout << "\ns4 = " << s4 << "\n\n";
57
58      // test overloaded copy assignment (=) operator with self-assignment
59      cout << "assigning s4 to s4" << endl;
60      s4 = s4;
61      cout << "s4 = " << s4 << endl;
62
63      // test using overloaded subscript operator to create lvalue
64      s1[ 0 ] = 'H';
65      s1[ 6 ] = 'B';
66      cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
67         << s1 << "\n\n";
68
```

Fig. 10.1 | Standard Library `string` class test program. (Part 3 of 6.)

```
69      // test subscript out of range with string member function "at"
70      try
71      {
72         cout << "Attempt to assign 'd' to s1.at( 30 ) yields:" << endl;
73         s1.at( 30 ) = 'd'; // ERROR: subscript out of range
74      } // end try
75      catch ( out_of_range &ex )
76      {
77         cout << "An exception occurred: " << ex.what() << endl;
78      } // end catch
79   } // end main
```

```
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
```

Fig. 10.1 | Standard Library `string` class test program. (Part 4 of 6.)

```
s2 >= s1 yields false
s2 <= s1 yields true

Testing s3.empty():
s3 is empty; assigning s1 to s3;
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
s1 = happy birthday to you

The substring of s1 starting at location 0 for
14 characters, s1.substr(0, 14), is:
happy birthday

The substring of s1 starting at
location 15, s1.substr(15), is:
to you
```

Fig. 10.1 | Standard Library `string` class test program. (Part 5 of 6.)

```
s4 = happy birthday to you

assigning s4 to s4
s4 = happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1.at( 30 ) yields:
An exception occurred: invalid string position
```

Fig. 10.1 | Standard Library `string` class test program. (Part 6 of 6.)

10.2 Using the Overloaded Operators of Standard Library Class `string` (cont.)

- Class `string`'s overloaded equality and relational operators perform lexicographical comparisons (i.e., like a dictionary ordering) using the numerical values of the characters (see Appendix B, ASCII Character Set) in each `string`.
- Class `string` provides member function `empty` to determine whether a `string` is empty, which we demonstrate in line 27.
  - Returns `true` if the `string` is empty; otherwise, it returns `false`.
- Line 36 demonstrates class `string`'s overloaded `+=` operator for string concatenation.
  - Line 41 demonstrates that a string literal can be appended to a `string` object by using operator `+=`. Line 42 displays the result.

10.2 Using the Overloaded Operators of Standard Library Class `string` (cont.)

- Class `string` provides member function `substr` (lines 47 and 52) to return a *portion* of a string as a `string` object.
  - The call to `substr` in line 47 obtains a 14-character substring (specified by the second argument) of `s1` starting at position 0 (specified by the first argument).
  - The call to `substr` in line 52 obtains a substring starting from position 15 of `s1`.
  - When the second argument is not specified, `substr` returns the *remainder* of the `string` on which it's called.
- Lines 64-65 use class `string`'s overloaded `[ ]` operator can create *lvalues* that enable new characters to replace existing characters in `s1`.
  - *Class `string`'s overloaded `[ ]` operator does not perform any bounds checking.*

10.2 Using the Overloaded Operators of Standard Library Class `string` (cont.)

- Class `string` *does* provide bounds checking in its member function `at`, which throws an exception if its argument is an invalid subscript.
  - If the subscript is valid, function `at` returns the character at the specified location as a modifiable *lvalue* or a nonmodifiable *lvalue* (e.g., a `const` reference), depending on the context in which the call appears.

10.3 Fundamentals of Operator Overloading

- As you saw in Fig. 10.1, operators provide a concise notation for manipulating string objects.
- You can use operators with your own user-defined types as well.
- Although C++ does not allow new operators to be created, it does allow most existing operators to be overloaded so that, when they're used with objects, they have meaning appropriate to those objects.

10.3 Fundamentals of Operator Overloading (cont.)

- Operator overloading is not automatic—you must write operator-overloading functions to perform the desired operations.
- An operator is overloaded by writing a non-`static` member function definition or non-member function definition as you normally would, except that the function name starts with the keyword `operator` followed by the symbol for the operator being overloaded.
  - For example, the function name `operator+` would be used to overload the addition operator (`+`) for use with objects of a particular class (or `enum`).

10.3 Fundamentals of Operator Overloading (cont.)

- When operators are overloaded as member functions, they must be non-`static`, because *they must be called on an object of the class* and operate on that object.
- To use an operator on class objects, you must define overloaded operator functions for that class—with three exceptions.
  - The *assignment operator (=)* may be used with *most* classes to perform *memberwise assignment* of the data members—each data member is assigned from the assignment's "source" object (on the right) to the "target" object (on the left).
    - *Memberwise assignment is dangerous for classes with pointer members*, so we'll explicitly overload the assignment operator for such classes.
  - The *address operator (&)* returns a pointer to the object; this operator also can be overloaded.
  - The *comma operator* evaluates the expression to its left then the expression to its right, and returns the value of the latter expression.

10.3 Fundamentals of Operator Overloading (cont.)

- Most of C++'s operators can be overloaded.
- Figure 10.2 shows the operators that cannot be overloaded.

Operators that cannot be overloaded			
.	.* (pointer to member)	::	?:

Fig. 10.2 | Operators that cannot be overloaded.

10.3 Fundamentals of Operator Overloading (cont.)

- *The precedence of an operator cannot be changed by overloading*.
  - However, parentheses can be used to force the order of evaluation of overloaded operators in an expression.
- *The associativity of an operator cannot be changed by overloading*
  - if an operator normally associates from left to right, then so do all of its overloaded versions.
- *You cannot change the "arity" of an operator* (that is, the number of operands an operator takes)
  - overloaded unary operators remain unary operators; overloaded binary operators remain binary operators. Operators &, *, + and - all have both unary and binary versions; these unary and binary versions can be separately overloaded.

10.3 Fundamentals of Operator Overloading (cont.)

- *You cannot create new operators; only existing operators can be overloaded.*
- The meaning of how an operator works on values of fundamental types *cannot* be changed by operator overloading.
  - For example, you cannot make the + operator subtract two `int`s.
- Operator overloading works only with *objects of user-defined types or with a mixture of an object of a user-defined type and an object of a fundamental type.*

10.3 Fundamentals of Operator Overloading (cont.)

- Related operators, like + and +=, must be overloaded separately.
- When overloading ( ), [ ], -> or any of the assignment operators, the operator overloading function must be declared as a class member.
  - For all other overloadable operators, the operator overloading functions can be member functions or non-member functions.

Software Engineering Observation 10.1

Overload operators for class types so they work as closely as possible to the way built-in operators work on fundamental types.

10.4 Overloading Binary Operators

- A binary operator can be overloaded as
  - a non-`static` member function with one parameter or as
  - a non-member function with two parameters (one of those parameters must be either a class object or a reference to a class object).
- As a non-member function, binary operator must take two arguments—one of which must be an object (or a reference to an object) of the class.

10.5 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.
- The program of Figs. 10.3–10.5 overloads these operators to input and output `PhoneNumber` objects in the format "`(000) 000-0000`." The program assumes telephone numbers are input correctly.

```
1  // Fig. 10.3: PhoneNumber.h
2  // PhoneNumber class definition
3  #ifndef PHONENUMBER_H
4  #define PHONENUMBER_H
5
6  #include <iostream>
7  #include <string>
8
9  class PhoneNumber
10 {
11    friend std::ostream &operator<<( std::ostream &, const PhoneNumber & );
12    friend std::istream &operator>>( std::istream &, PhoneNumber & );
13 private:
14    std::string areaCode; // 3-digit area code
15    std::string exchange; // 3-digit exchange
16    std::string line; // 4-digit line
17 }; // end class PhoneNumber
18
19 #endif
```

Fig. 10.3 | PhoneNumber class with overloaded stream insertion and stream extraction operators as `friend` functions.

```
1  // Fig. 10.4: PhoneNumber.cpp
2  // Overloaded stream insertion and stream extraction operators
3  // for class PhoneNumber.
4  #include <iomanip>
5  #include "PhoneNumber.h"
6  using namespace std;
7
8  // overloaded stream insertion operator; cannot be
9  // a member function if we would like to invoke it with
10 // cout << somePhoneNumber;
11 ostream &operator<<( ostream &output, const PhoneNumber &number )
12 {
13    output << "(" << number.areaCode << ") "
14       << number.exchange << "-" << number.line;
15    return output; // enables cout << a << b << c;
16 } // end function operator<<
17
```

Fig. 10.4 | Overloaded stream insertion and stream extraction operators for class PhoneNumber. (Part 1 of 2.)

```
18  // overloaded stream extraction operator; cannot be
19  // a member function if we would like to invoke it with
20  // cin >> somePhoneNumber;
21  istream &operator>>( istream &input, PhoneNumber &number )
22  {
23     input.ignore(); // skip (
24     input >> setw( 3 ) >> number.areaCode; // input area code
25     input.ignore( 2 ); // skip ) and space
26     input >> setw( 3 ) >> number.exchange; // input exchange
27     input.ignore(); // skip dash (-)
28     input >> setw( 4 ) >> number.line; // input line
29     return input; // enables cin >> a >> b >> c;
30  } // end function operator>>
```

Fig. 10.4 | Overloaded stream insertion and stream extraction operators for class PhoneNumber. (Part 2 of 2.)

```
1  // Fig. 10.5: fig10_05.cpp
2  // Demonstrating class PhoneNumber's overloaded stream insertion
3  // and stream extraction operators.
4  #include <iostream>
5  #include "PhoneNumber.h"
6  using namespace std;
7
8  int main()
9  {
10    PhoneNumber phone; // create object phone
11
12    cout << "Enter phone number in the form (123) 456-7890:" << endl;
13
14    // cin >> phone invokes operator>> by implicitly issuing
15    // the non-member function call operator>>( cin, phone )
16    cin >> phone;
17
18    cout << "The phone number entered was: ";
19
20    // cout << phone invokes operator<< by implicitly issuing
21    // the non-member function call operator<<( cout, phone )
22    cout << phone << endl;
23 } // end main
```

Fig. 10.5 | Overloaded stream insertion and stream extraction operators. (Part 1 of 2.)

```
Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
```

Fig. 10.5 | Overloaded stream insertion and stream extraction operators. (Part 2 of 2.)

10.5 Overloading the Binary Stream Insertion and Stream Extraction Operators (cont.)

Overloading the Stream Extraction (>>) Operator

- The stream extraction operator function `operator>>` (Fig. 10.4, lines 21–30) takes `istream` reference `input` and `PhoneNumber` reference `number` as arguments and returns an `istream` reference.
- Operator function `operator>>` inputs phone numbers of the form
  - `(800) 555-1212`
- When the compiler sees the expression
  - `cin >> phone`
- In line 16 of Fig. 10.5, the compiler generates the *non-member function call*
  - `operator>>( cin, phone );`
- When this call executes, reference parameter `input` (Fig. 10.4, line 21) becomes an alias for `cin` and reference parameter `number` becomes an alias for `phone`.

10.5 Overloading the Binary Stream Insertion and Stream Extraction Operators (cont.)

- The operator function reads as `strings` the three parts of the telephone number into the `areaCode` (line 24), `exchange` (Line 26) and `line` (line 28) members of the `PhoneNumber` object referenced by parameter `Number`.
- Stream manipulator `setw` limits the number of characters read into each `string`.
- The parentheses, space and dash characters are skipped by calling `istream` member function `ignore` (Fig. 10.4, lines 23, 25 and 27), which discards the specified number of characters in the input stream (one character by default).

10.5 Overloading Stream Insertion and Stream Extraction Operators (cont.)

- Function `operator>>` returns `istream` reference `input` (i.e., `cin`).
- This enables input operations on `PhoneNumber` objects to be cascaded with input operations on other `PhoneNumber` objects or on objects of other data types.

Good Programming Practice 10.1

Overloaded operators should mimic the functionality of their built-in counterparts—e.g., the `+` operator should perform addition, not subtraction. Avoid excessive or inconsistent use of operator overloading, as this can make a program cryptic and difficult to read.

10.5 Overloading Stream Insertion and Stream Extraction Operators (cont.)

Overloading the Stream Insertion (<<) Operator

- The stream insertion operator function (Fig. 10.4, lines 11-16) takes an `ostream` reference (`output`) and a `const PhoneNumber` reference (`number`) as arguments and returns an `ostream` reference.
- Function `operator<<` displays objects of type `PhoneNumber`.
- When the compiler sees the expression
  - `cout << phone`

  in line 22 if Fig. 10.5, the compiler generates the non-member function call
  - `operator<<( cout, phone );`
- Function `operator<<` displays the parts of the telephone number as `strings`, because they're stored as `string` objects.

10.5 Overloading the Binary Stream Insertion and Stream Extraction Operators (cont.)

Overloaded Operators as Non-Member `friend` Functions

- The functions `operator>>` and `operator<<` are declared in `PhoneNumber` as non-member, `friend` functions.
- They're *non-member functions* because the object of class `PhoneNumber` is the operator's *right* operand.

Software Engineering Observation 10.2

New input/output capabilities for user-defined types are added to C++ without modifying standard input/output library classes. This is another example of C++'s extensibility.

10.5 Overloading the Binary Stream Insertion and Stream Extraction Operators (cont.)

Why Overloaded Stream Insertion and Stream Extraction Operators Are Overloaded as Non-Member Functions

- The overloaded stream insertion operator (`<<`) is used in an expression in which the left operand has type `ostream &`, as in `cout << classObject`.
- To use the operator in this manner where the *right* operand is an object of a user-defined class, it must be overloaded as a *non-member function*.

10.5 Overloading the Binary Stream Insertion and Stream Extraction Operators (cont.)

- Similarly, the overloaded stream extraction operator (`>>`) is used in an expression in which the left operand has type `istream &`, as in `cin >> classObject`, and the *right* operand is an object of a user-defined class, so it, too, must be a non-member function.
- Each of these overloaded operator functions may require access to the `private` data members of the class object being output or input, so these overloaded operator functions can be made `friend` functions of the class for performance reasons.

10.6 Overloading Unary Operators

- *A unary operator for a class can be overloaded as a non-`static` 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.*
- A unary operator such as ! may be overloaded as a *non-member function* with one parameter.

10.7 Overloading the Unary Prefix and Postfix ++ and -- Operators

- The prefix and postfix versions of the increment and decrement operators can all be overloaded.
- *To overload the increment operator to allow both prefix and postfix increment usage, each overloaded operator function must have a distinct signature, so that the compiler will be able to determine which version of ++ is intended.*
- The prefix versions are overloaded exactly as any other prefix unary operator would be.

10.7 Overloading the Unary Prefix and Postfix ++ and -- Operators (cont.)

- Suppose that we want to add 1 to the day in `Date` object `d1`.
- When the compiler sees the preincrementing expression `++d1`, the compiler generates the *member-function call*
  - `d1.operator++()`
- The prototype for this operator function would be
  - `Date &operator++();`
- If the prefix increment operator is implemented as a non-member function, then, when the compiler sees the expression `++d1`, the compiler generates the function call
  - `operator++( d1 )`
- The prototype for this operator function would be declared in the `Date` class as
  - `Date &operator++( Date & );`

10.7 Overloading the Unary Prefix and Postfix ++ and -- Operators (cont.)

Overloading the Postfix Increment Operator

- Overloading the postfix increment operator presents a challenge, because the compiler must be able to distinguish between the signatures of the overloaded prefix and postfix increment operator functions.
- The *convention* that has been adopted in C++ is that, when the compiler sees the postincrementing expression `d1++`, it generates the *member-function call*
  - `d1.operator++( 0 )`
- The prototype for this function is
  - `Date operator++( int )`
- The argument `0` is strictly a "dummy value" that enables the compiler to distinguish between the prefix and postfix increment operator functions.
- The same syntax is used to differentiate between the prefix and postfix decrement operator functions.

10.7 Overloading the Unary Prefix and Postfix ++ and -- Operators (cont.)

- If the postfix increment is implemented as a non-member function, then, when the compiler sees the expression `d1++`, the compiler generates the function call
  - `operator++( d1, 0 )`
- The prototype for this function would be
  - `Date operator++( Date &, int );`
- Once again, the `0` argument is used by the compiler to distinguish between the prefix and postfix increment operators implemented as non-member functions.
- The postfix increment operator returns `Date` objects *by value*, whereas the prefix increment operator returns `Date` objects *by reference*—the postfix increment operator typically returns a temporary object that contains the original value of the object before the increment occurred.

Performance Tip 10.1

The extra object that's created by the postfix increment (or decrement) operator can result in a performance problem—especially when the operator is used in a loop. For this reason, you should prefer the overloaded prefix increment and decrement operators.

10.8 Case Study: A `Date` Class

- The program of Figs. 10.6–10.8 demonstrates a `Date` class, which uses overloaded prefix and postfix increment operators to add 1 to the day in a `Date` object, while causing appropriate increments to the month and year if necessary.

```
1  // Fig. 10.6: Date.h
2  // Date class definition with overloaded increment operators.
3  #ifndef DATE_H
4  #define DATE_H
5
6  #include <array>
7  #include <iostream>
8
9  class Date
10 {
11    friend std::ostream &operator<<( std::ostream &, const Date & );
12 public:
13    Date( int m = 1, int d = 1, int y = 1900 ); // default constructor
14    void setDate( int, int, int ); // set month, day, year
15    Date &operator++(); // prefix increment operator
16    Date operator++( int ); // postfix increment operator
17    Date &operator+=( unsigned int ); // add days, modify object
18    static bool leapYear( int ); // is date in a leap year?
19    bool endOfMonth( int ) const; // is date at the end of month?
20 private:
21    unsigned int month;
22    unsigned int day;
23    unsigned int year;
```

Fig. 10.6 | `Date` class definition with overloaded increment operators. (Part 1 of 2.)

```
24
25      static const std::array< unsigned int, 13 > days; // days per month
26      void helpIncrement(); // utility function for incrementing date
27   }; // end class Date
28
29   #endif
```

Fig. 10.6 | `Date` class definition with overloaded increment operators. (Part 2 of 2.)

```
1  // Fig. 10.7: Date.cpp
2  // Date class member- and friend-function definitions.
3  #include <iostream>
4  #include <string>
5  #include "Date.h"
6  using namespace std;
7
8  // initialize static member; one classwide copy
9  const array< unsigned int, 13 > Date::days =
10    { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
11
12 // Date constructor
13 Date::Date( int month, int day, int year )
14 {
15    setDate( month, day, year );
16 } // end Date constructor
17
```

Fig. 10.7 | Date class member- and friend-function definitions. (Part 1 of 6.)

```
18  // set month, day and year
19  void Date::setDate( int mm, int dd, int yy )
20  {
21     if ( mm >= 1 && mm <= 12 )
22        month = mm;
23     else
24        throw invalid_argument( "Month must be 1-12" );
25
26     if ( yy >= 1900 && yy <= 2100 )
27        year = yy;
28     else
29        throw invalid_argument( "Year must be >= 1900 and <= 2100" );
30
31     // test for a leap year
32     if ( ( month == 2 && leapYear( year ) && dd >= 1 && dd <= 29 ) ||
33          ( dd >= 1 && dd <= days[ month ] ) )
34        day = dd;
35     else
36        throw invalid_argument(
37           "Day is out of range for current month and year" );
38  } // end function setDate
39
```

Fig. 10.7 | `Date` class member- and `friend`-function definitions. (Part 2 of 6.)

```
40 // overloaded prefix increment operator
41 Date &Date::operator++()
42 {
43    helpIncrement(); // increment date
44    return *this; // reference return to create an lvalue
45 } // end function operator++
46
47 // overloaded postfix increment operator; note that the
48 // dummy integer parameter does not have a parameter name
49 Date Date::operator++( int )
50 {
51    Date temp = *this; // hold current state of object
52    helpIncrement();
53
54    // return unincremented, saved, temporary object
55    return temp; // value return; not a reference return
56 } // end function operator++
57
```

Fig. 10.7 | `Date` class member- and `friend`-function definitions. (Part 3 of 6.)

```
58  // add specified number of days to date
59  Date &Date::operator+=( unsigned int additionalDays )
60  {
61     for ( int i = 0; i < additionalDays; ++i )
62        helpIncrement();
63
64     return *this; // enables cascading
65  } // end function operator+=
66
67  // if the year is a leap year, return true; otherwise, return false
68  bool Date::leapYear( int testYear )
69  {
70     if ( testYear % 400 == 0 ||
71        ( testYear % 100 != 0 && testYear % 4 == 0 ) )
72        return true; // a leap year
73     else
74        return false; // not a leap year
75  } // end function leapYear
76
```

Fig. 10.7 | `Date` class member- and `friend`-function definitions. (Part 4 of 6.)

```
77  // determine whether the day is the last day of the month
78  bool Date::endOfMonth( int testDay ) const
79  {
80     if ( month == 2 && leapYear( year ) )
81        return testDay == 29; // last day of Feb. in leap year
82     else
83        return testDay == days[ month ];
84  } // end function endOfMonth
85
86  // function to help increment the date
87  void Date::helpIncrement()
88  {
89     // day is not end of month
90     if ( !endOfMonth( day ) )
91        ++day; // increment day
92     else
93        if ( month < 12 ) // day is end of month and month < 12
94        {
95           ++month; // increment month
96           day = 1; // first day of new month
97        } // end if
```

Fig. 10.7 | `Date` class member- and `friend`-function definitions. (Part 5 of 6.)

```
98         else // last day of year
99         {
100           ++year; // increment year
101           month = 1; // first month of new year
102           day = 1; // first day of new month
103        } // end else
104  } // end function helpIncrement
105
106  // overloaded output operator
107  ostream &operator<<( ostream &output, const Date &d )
108  {
109     static string monthName[ 13 ] = { "", "January", "February",
110        "March", "April", "May", "June", "July", "August",
111        "September", "October", "November", "December" };
112     output << monthName[ d.month ] << ' ' << d.day << ", " << d.year;
113     return output; // enables cascading
114  } // end function operator<<
```

Fig. 10.7 | `Date` class member- and `friend`-function definitions. (Part 6 of 6.)

```
1  // Fig. 10.8: fig10_08.cpp
2  // Date class test program.
3  #include <iostream>
4  #include "Date.h" // Date class definition
5  using namespace std;
6
7  int main()
8  {
9     Date d1( 12, 27, 2010 ); // December 27, 2010
10    Date d2; // defaults to January 1, 1900
11
12    cout << "d1 is " << d1 << "\nd2 is " << d2;
13    cout << "\n\nd1 += 7 is " << ( d1 += 7 );
14
15    d2.setDate( 2, 28, 2008 );
16    cout << "\n\n  d2 is " << d2;
17    cout << "\n++d2 is " << ++d2 << " (leap year allows 29th)";
18
19    Date d3( 7, 13, 2010 );
20
21    cout << "\n\nTesting the prefix increment operator:\n"
22       << "  d3 is " << d3 << endl;
23    cout << "++d3 is " << ++d3 << endl;
24    cout << "  d3 is " << d3;
```

Fig. 10.8 | Date class test program. (Part 1 of 2.)

```
25
26      cout << "\n\nTesting the postfix increment operator:\n"
27         << "  d3 is " << d3 << endl;
28      cout << "d3++ is " << d3++ << endl;
29      cout << "  d3 is " << d3 << endl;
30   } // end main
```

```
d1 is December 27, 2010
d2 is January 1, 1900

d1 += 7 is January 3, 2011

  d2 is February 28, 2008
++d2 is February 29, 2008 (leap year allows 29th)

Testing the prefix increment operator:
  d3 is July 13, 2010
++d3 is July 14, 2010
  d3 is July 14, 2010

Testing the postfix increment operator:
  d3 is July 14, 2010
d3++ is July 14, 2010
  d3 is July 15, 2010
```

Fig. 10.8 | `Date` class test program. (Part 2 of 2.)

10.8 Case Study: A `Date` Class (cont.)

- The `Date` constructor (defined in Fig. 10.7, lines 13–16) calls `setDate` to validate the month, day and year specified.
  - Invalid values for the month, day or year result in `invalid_argument` exceptions.

10.8 Case Study: A `Date` Class (cont.)

Date Class Prefix Increment Operator

- Overloading the prefix increment operator is straightforward.
  - The prefix increment operator (defined in Fig. 10.7, lines 41–45) calls utility function `helpIncrement` (defined in Fig. 10.7, lines 87–104) to increment the date.
  - This function deals with "wraparounds" or "carries" that occur when we increment the last day of the month.
  - These carries require incrementing the month.
  - If the month is already 12, then the year must also be incremented and the month must be set to 1.
  - Function `helpIncrement` uses function `endOfMonth` to determine whether the end of a month has been reached and increment the day correctly.

10.8 Case Study: A `Date` Class (cont.)

- The overloaded prefix increment operator returns a reference to the current `Date` object (i.e., the one that was just incremented).
- This occurs because the current object, `*this`, is returned as a `Date &`.
  - Enables a preincremented `Date` object to be used as an *lvalue,* which is how the built-in prefix increment operator works for fundamental types.

10.8 Case Study: A `Date` Class (cont.)

Date Class Postfix Increment Operator

- Overloading the postfix increment operator (defined in Fig. 10.7, lines 49–56) is trickier.
- To emulate the effect of the postincrement, we must return an unincremented copy of the `Date` object.
- So we'd like our postfix increment operator to operate the same way on a `Date` object.
- On entry to `operator++`, we save the current object (`*this`) in `temp` (line 51).
- Next, we call `helpIncrement` to increment the current `Date` object.
- Then, line 55 returns the unincremented copy of the object previously stored in `temp`.
- This function cannot return a reference to the local `Date` object `temp`, because a local variable is destroyed when the function in which it's declared exits.

Common Programming Error 10.1

Returning a reference (or a pointer) to a local variable is a common error for which most compilers will issue a warning.

10.9 Dynamic Memory Management

- You can control the *allocation* and *deallocation* of memory in a program for objects and for arrays of any built-in or user-defined type.
  - Known as dynamic memory management; performed with `new` and `delete`.
- You can use the `new` operator to dynamically allocate (i.e., reserve) the exact amount of memory required to hold an object or built-in array at execution time.
- The object or built-in array is created in the free store (also called the heap)—*a region of memory assigned to each program for storing dynamically allocated objects.*
- Once memory is allocated in the free store, you can access it via the pointer that operator `new` returns.
- You can return memory to the free store by using the `delete` operator to deallocate it.

10.9 Dynamic Memory Management (cont.)

Obtaining Dynamic Memory with new

- The `new` operator allocates storage of the proper size for an object of type `Time`, calls the default constructor to initialize the object and returns a pointer to the type specified to the right of the `new` operator (i.e., a `Time *`).
- If `new` is unable to find sufficient space in memory for the object, it indicates that an error occurred by "throwing an exception."

10.9 Dynamic Memory Management (cont.)

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.*

Common Programming Error 10.2

Not releasing dynamically allocated memory when it's no longer needed can cause the system to run out of memory prematurely. This is sometimes called a "memory leak."

Error-Prevention Tip 10.1

Do not delete memory that was not allocated by `new`. Doing so results in undefined behavior.

Error-Prevention Tip 10.2

After you delete a block of dynamically allocated memory be sure not to delete the same block again. One way to guard against this is to immediately set the pointer to `nullptr`. Deleting a `nullptr` has no effect.

10.9 Dynamic Memory Management (cont.)

Initializing Dynamic Memory

- You can provide an initializer for a newly created fundamental-type variable, as in
  - `double *ptr = new double( 3.14159 );`
- The same syntax can be used to specify a comma-separated list of arguments to the constructor of an object.

10.9 Dynamic Memory Management (cont.)

Dynamically Allocating Built-In Arrays with new [ ]

- You can also use the `new` operator to allocate built-in arrays dynamically.
- For example, a 10-element integer array can be allocated and assigned to `gradesArray` as follows:
  - `int *gradesArray = new int[ 10 ]();`
- The parentheses following new `int[10]` value initialize the array's elements—fundamental numeric types are set to `0`, `bool`s are set to `false`, pointers are set to `nullptr` and class objects are initialized by their default constructors.
- A dynamically allocated array's size can be specified using *any* non-negative integral expression that can be evaluated at execution time.

10.9 Dynamic Memory Management (cont.)

C++11: Using a List Initializer with a Dynamically Allocated Built-In Array

- Prior to C++11, when allocating a built-in array of objects dynamically, you could not pass arguments to each object's constructor—each object was initialized by its default constructor. In C++11, you can use a list initializer to initialize the elements of a dynamically allocated built-in array, as in

```
int *gradesArray = new int[ 10 ]{};
```

- The empty set of braces as shown here indicates that default initialization should be used for each element—for fundamental types each element is set to 0.
- The braces may also contain a comma-separated list of initializers for the array's elements.

10.9 Dynamic Memory Management (cont.)

Releasing Dynamically Allocated Built-In Arrays with delete [ ]

- To deallocate a dynamically allocated array, use the statement
  - `delete [] ptr;`
- *If the pointer points to a built-in array of objects, the statement first calls the destructor for every object in the array, then deallocates the memory.*
- *Using `delete` or `[ ]` on a `nullptr` has no effect.*

Common Programming Error 10.3

Using `delete` instead of `delete []` for built-in arrays of objects can lead to runtime logic errors. To ensure that every object in the array receives a destructor call, always delete memory allocated as an array with operator `delete []`. Similarly, always delete memory allocated as an individual element with operator `delete`—the result of deleting a single object with operator `delete []` is undefined.

10.9 Dynamic Memory Management (cont.)

C++11: Managing Dynamically Allocated Memory with `unique_ptr`

- C++11's new `unique_ptr` is a "smart pointer" for managing dynamically allocated memory.
- When a `unique_ptr` goes out of scope, its destructor automatically returns the managed memory to the free store.

10.10 Case Study: `Array` Class

- Pointer-based arrays have many problems, including:
  - A program can easily "walk off" either end of a built-in array, because *C++ does not check whether subscripts fall outside the range of the array.*
  - Built-in arrays of size *n* must number their elements 0, …, *n* – 1; alternate subscript ranges- are not allowed.
  - An entire built-in array cannot be input or output at once.
  - Two built-in arrays cannot be meaningfully compared with equality or relational operators.
  - When an array is passed to a general-purpose function designed to handle arrays of any size, the array's size must be passed as an additional argument.
  - One built-in array cannot be assigned to another with the assignment operator.

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

- With C++, you can implement more robust array capabilities via classes and operator overloading as has been done with class templates `array` and `vector` in the C++ Standard Library.
- 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 `!=`.
- C++ Standard Library class template `vector` provides many of these capabilities as well.

```
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.)

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

- The `Array` copy constructor copies the elements of one `Array` into another.
- The copy constructor can also be invoked by writing line 39 as follows:
  - `Array integers3 = integers1;`
- 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.
- This form can be used to pass only a single argument to a constructor.

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

- *The array subscript operator [ ] is not restricted for use only with arrays*; it also can be used, for example, to select elements from other kinds of *container classes*, such as strings and dictionaries.
- Also, when `operator[]` functions are defined, *subscripts no longer have to be integers*—characters, strings, floats or even objects of user-defined classes also could be used.

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

- Each `Array` object consists of a `size` member indicating the number of elements in the `Array` and an `int` pointer—`ptr`—that points to the dynamically allocated pointer-based array of integers managed by the `Array` object.
- When the compiler sees an expression like `cout << arrayObject`, it invokes non-member function `operator<<` with the call
  - `operator<<( cout, arrayObject )`
- When the compiler sees an expression like `cin >> arrayObject`, it invokes non-member function `operator>>` with the call
  - `operator>>( cin, arrayObject )`
- These stream insertion and stream extraction operator functions cannot be members of class `Array`, because the `Array` object is always mentioned on the right side of the stream insertion operator and the stream extraction operator.

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

- You might be tempted to replace the counter-controlled `for` statement in lines 104–105 and many of the other `for` statements in class `Array`'s implementation with the C++11 range-based for statement.
- Unfortunately, range-based `for` does *not* work with dynamically allocated built-in arrays.

```
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.)

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 `Arrays` 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.

Common Programming Error 10.5

Not providing a copy constructor and overloaded assignment operator for a class when objects of that class contain pointers to dynamically allocated memory is a potential logic error.

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

C++11: Deleting Unwanted Member Functions from Your Class

- Prior to C++11, you could prevent class objects from being *copied* or *assigned* by declaring as `private` the class's copy constructor and overloaded assignment operator.
- As of C++11, you can simply *delete* these functions from your class.
- To do so in class `Array`, replace the prototypes in lines 15 and 19 of Fig. 10.10 with:

```
Array( const Array & ) = delete;
const Array &operator=( const Array & ) = delete;
```

- Though you can delete any member function, it's most commonly used with member functions that the compiler can auto-generate—the default constructor, copy constructor, assignment operator, and in C++ 11, the move constructor and move assignment operator.

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

Overloaded Equality and Inequality Operators

- Line 20 of Fig. 10.10 declares the overloaded equality operator (`==`) for the class.
- When the compiler sees the expression `integers1 == integers2` in line 55 of Fig. 10.9, the compiler invokes member function `operator==` with the call
  - `integers1.operator==( integers2 )`
- Member function `operator==` (defined in Fig. 10.11, lines 66–76) immediately returns `false` if the `size` members of the `Array`s are not equal.
- Otherwise, `operator==` compares each pair of elements.
- If they're all equal, the function returns `true`.
- The first pair of elements to differ causes the function to return `false` immediately.

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

- Lines 23–26 Fig. 10.9 define the overloaded inequality operator (`!=`) for the class.
- Member function `operator!=` uses the overloaded `operator==` function to deter-mine whether one `Array` is equal to another, then returns the opposite of that result.
- Writing `operator!=` in this manner enables you to reuse `operator==`, which *reduces the amount of code that must be written in the class*.
- Also, the full function definition for `operator!=` is in the `Array` header.
  - Allows the compiler to inline the definition.

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

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).
- When the compiler sees the expression `integers1[5]` (Fig. 10.9, line 59), it invokes the appropriate overloaded `operator[]` member function by generating the call
  - `integers1.operator[]( 5 )`
- 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.

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

- Each definition of `operator[]` determines whether the subscript it receives as an argument is in range.
- If it isn't, each function prints an error message and terminates the program with a call to function `exit`.
- If the subscript is in range, the non-`const` version of `operator[]` returns the appropriate `Array` element as a reference so that it may be used as a modifiable *lvalue.*
- If the subscript is in range, the `const` version of `operator[]` returns a copy of the appropriate element of the `Array`.

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

C++11: Managing Dynamically Allocated Memory with `unique_ptr`

- In this case study, class `Array`'s destructor used `delete []` to return the dynamically allocated built-in array to the free store.
- As you recall, C++11 enables you to use `unique_ptr` to ensure that this dynamically allocated memory is deleted when the `Array` object goes out of scope.

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

C++11: Passing a List Initializer to a Constructor

- In Fig. 7.4, we showed how to initialize an array object with a comma-separated list of initializers in braces, as in

```
array< int, 5 > n = { 32, 27, 64, 18, 95 };
```

- C++11 now allows any object to be initialized with a list initializer and that the preceding statement can also be written without the =, as in

```
array< int, 5 > n{ 32, 27, 64, 18, 95 };
```

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

- C++11 also allows you to use list initializers when you declare objects of your own classes.
- For example, you can now provide an Array constructor that would enabled the following declarations:

```
Array integers = { 1, 2, 3, 4, 5 };
```

- or

```
Array integers{ 1, 2, 3, 4, 5 };
```

- each of which creates an `Array` object with five elements containing the integers from 1 to 5.

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

- To support list initialization, you can define a constructor that receives an *object* of the class template `initializer_list`. For class `Array`, you'd include the `<initializer_list>` header.
- Then, you'd define a constructor with the first line:

```
Array::Array( initializer_list< int > list )
```

- You can determine the number of elements in the list parameter by calling its size member function.
- To obtain each initializer and copy it into the `Array` object's dynamically allocated built-in array, you can use a range-based for as follows:

```
size_t i = 0;
for ( int item : list )
   ptr[ i++ ] = item;
```

10.11 Operators as Member vs. Non-Member Functions

- Whether an operator function is implemented as a *member function* or as a non-member function, the operator is still used the same way in expressions.
- 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 (as we did in Section 10.5 when overloading `<<` and `>>` as the stream insertion and extraction operators, respectively).
- A non-member operator function can be made a `friend` of a class if that function must access `private` or `protected` members of that class directly.
- Operator member functions of a specific class are called only when the left operand of a binary operator is specifically an object of that class, or when the *single operand of a unary operator* is an object of that class.

10.11 Operators as Member vs. Non-Member Functions (cont.)

- You might choose a non-member function to overload an operator to enable the operator to be *commutative*.
- The `operator+` function that deals with the `HugeInt` on the left, can still be a *member function*.
- The *non-member function* simply swaps its arguments and calls the *member function*.

10.12 Converting Between Types

- Sometimes all the operations "stay within a type." For example, adding an `int` to an `int` produces an `int`.
- It's often necessary, however, to convert data of one type to data of another type.
- The compiler knows how to perform certain conversions among fundamental types.
- You can use *cast operators* to *force* conversions among fundamental types.
- The compiler cannot know in advance how to convert among user-defined types, and between user-defined types and fundamental types, so you must specify how to do this.

10.12 Converting Between Types (cont.)

- Such conversions can be performed with conversion constructors—constructors that can be called with a single argument (we'll refer to these as *single-argument constructors*).
- Such constructors can turn objects of other types (including fundamental types) into objects of a particular class.

10.12 Converting Between Types (cont.)

Conversion Operators

- A conversion operator (also called a *cast operator*) can be used to convert an object of one class to another type.
- Such a conversion operator must be a *non-`static` member function.*
- The function prototype
  - `MyClass::operator char *() const;`
- declares an overloaded cast operator function for converting an object of class `MyClass` into a temporary `char *` object.
- The operator function is declared `const` because it does *not* modify the original object.

10.12 Converting between Types (cont.)

- The return type of an overloaded cast operator function is implicitly the type to which the object is being converted.
- If `s` is a class object, when the compiler sees the expression `static_cast< char * >( s )`, the compiler generates the call
  - `s.operator char *()`

10.12 Converting between Types (cont.)

Overloaded Cast Operator Functions

- Overloaded cast operator functions can be defined to convert objects of user-defined types into fundamental types or into objects of other user-defined types.

Implicit Calls to Cast Operators and Conversion Constructors

- One of the nice features of cast operators and conversion constructors is that, when necessary, the compiler can call these functions *implicitly* to create *temporary objects*.

Software Engineering Observation 10.5

When a conversion constructor or conversion operator is used to perform an implicit conversion, C++ can apply only one implicit constructor or operator function call (i.e., a single user-defined conversion) to try to match the needs of another overloaded operator. The compiler will not satisfy an overloaded operator's needs by performing a series of implicit, user-defined conversions.

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.*
- For example, our `Array` class in Fig. 10.10 defines a constructor that takes a single `int` argument.
- The intent of this constructor is to create an `Array` object containing the number of elements specified by the `int` argument.
- However, if this constructor were not declared `explicit` it could be misused by the compiler to perform an *implicit conversion*.

Common Programming Error 10.6

Unfortunately, the compiler might use implicit conversions in cases that you do not expect, resulting in ambiguous expressions that generate compilation errors or result in execution-time logic errors.

10.13 `explicit` Constructors and Conversion Operators (cont.)

- The program (Fig. 10.12) uses the `Array` class of Figs. 10.10–10.11 to demonstrate an improper implicit conversion.
- Line 13 calls function `outputArray` with the `int` value `3` as an argument.
- This program does not contain a function called `outputArray` that takes an `int` argument.
  - The compiler determines whether class `Array` provides a conversion constructor that can convert an `int` into an `Array`.
  - The compiler assumes the `Array` constructor that receives a single `int` is a conversion constructor and uses it to convert the argument `3` into a temporary `Array` object that contains three elements.
  - Then, the compiler passes the temporary `Array` object to function `outputArray` to output the `Array`'s contents.

```
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.)

```
The Array received has 7 elements. The contents are:
           0           0           0           0
           0           0           0

The Array received has 3 elements. The contents are:
           0           0           0
```

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

10.13 `explicit` Constructors and Conversion Operators (cont.)

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.*
- A constructor that is declared `explicit` cannot be used in an implicit conversion.
- In the example if Figure 10.13, we use the original version of `Array.h` from Fig. 10.10, which included the keyword `explicit` in the declaration of the *single-argument constructor* in line 14.

10.13 `explicit` Constructors and Conversion Operators (cont.)

- Figure 10.13 presents a slightly modified version of the program in Fig. 10.12.
- When this program is compiled, the compiler produces an error message indicating that the integer value passed to `outputArray` in line 13 cannot be converted to a `const Array &`.
- The compiler error message (from Visual C++) is shown in the output window.
- Line 14 demonstrates how the explicit constructor can be used to create a temporary `Array` of 3 elements and pass it to function `outputArray`.

Error-Prevention Tip 10.4

Always use the `explicit` keyword on single-argument constructors unless they're intended to be used as conversion constructors.

```
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.)

10.13 `explicit` Constructors and Conversion Operators (cont.)

C++11: `explicit` Conversion Operators

- As of C++11, 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`.

10.14 Overloading the Function Call Operator ( )

- Overloading the function call operator ( ) is powerful, because functions can take an arbitrary number of comma-separated parameters.
- In a *customized* `String` class, for example, you could overload this operator to select a substring from a `String`—the operator's two integer parameters could specify the *start location* and the *length of the substring to be selected.*
- The `operator()` function could check for such errors as a *start location out of range* or a *negative substring length*.
- The overloaded function call operator must be a non-static member function and could be defined with the first line:

```
String String::operator()( size_t index, size_t length ) const
```

10.14 Overloading the Function Call Operator ( )

- In this case, it should be a `const` member function because obtaining a substring should *not* modify the original String object.
- Suppose `string1` is a String object containing the string `"AEIOU"`.
- When the compiler encounters the expression `string1(2, 3)`, it generates the member-function call

```
string1.operator()( 2, 3 )
```

- which returns a `String` containing `"IOU"`.
- Another possible use of the function call operator is to enable an alternate `Array` subscripting notation.
- Instead of using C++'s double-square-bracket notation, such as in `chessBoard[row][column]`, you might prefer to overload the function call operator to enable the notation `chessBoard(row, column)`, where `chessBoard` is an object of a modified two-dimensional `Array` class.