Computing III

91.201 Spring 2016

Lecture – Operator overloading

Prof. Anna Rumshisky
Dept. of Computer Science
University of Massachusetts, Lowell

Operator Overloading

11.3 Fundamentals of Operator Overloading

- Operators provide a concise notation for manipulating STL objects.
- Can be used with user-defined types.
- 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.

11.3 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.

11.3 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, that operator must be overloaded—with three exceptions.
  - The assignment operator (`=`) may be used with *every* class to perform *memberwise assignment* of the class's 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.

Operators that can not be overloaded

- Member accessor operators “.” and “::”
- Ternary operator “?:”
- `sizeof` operator
- Pointer to member operators “.*”

Pointer-to-Member Operators

```
class Testpm {
public:
   void m_func1() { cout << "m_func1\n"; }
   int m_num;
};

// Define pointers to member function and data member
void (Testpm::*pmfn)() = &Testpm::m_func1;
int Testpm::*pmd = &Testpm::m_num;

int main() {
   Testpm ATestpm;
   Testpm *pTestpm = new Testpm;

// Access the member function
   (ATestpm.*pmfn)();
   (pTestpm->*pmfn)();   // Parentheses required, or function call () would bind first

// Access the member data
   ATestpm.*pmd = 1;
   pTestpm->*pmd = 2;
```

Operators that can be overloaded

- Almost all
  - arithmetic operators: + - * / % and += -= *= /= %= (all binary infix); + - (unary prefix); ++ -- (unary prefix and postfix)
  - bit manipulation: & | ^ << >> and &= |= ^= <<= >>= (all binary infix); ~ (unary prefix)
  - boolean algebra: == != < > <= >= || && (all binary infix); ! (unary prefix)
  - memory management: new new[] delete delete[]
  - implicit conversion operators
  - miscellany: = [] -> , (all binary infix); * & (all unary prefix) () (function call, n-ary infix)

11.3 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.

11.3 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 ints. 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.

No redefining built-in data types!

- The meaning of an operator for the built-in types may not be changed
  - For example, the built-in integer addition operation cannot be redefined:

```
// error:

int operator+(int, int);
```

- May not define additional operators for the built-in data types
  - For example, an operator+ taking two operands of array types cannot be defined.

11.3 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.

11.4 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.

Performance Tip 11.1

It's possible to overload an operator as a non-member, non-`friend` function, but such a function requiring access to a class's `private` or `protected` data would need to use set or get functions provided in that class's `public` interface. The overhead of calling these functions could cause poor performance, so these functions can be inlined to improve performance.

11.5 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.
- Let's overload these operators to input and output PhoneNumber objects in the format “(000) 000-0000.” We will assume the telephone numbers are input correctly.

```
1  // Fig. 11.3: PhoneNumber.h
2  // PhoneNumber class definition
3  #ifndef PHONENUMBER_H
4  #define PHONENUMBER_H
5
6  #include <iostream>
7  #include <string>
8  using namespace std;
9
10 class PhoneNumber
11 {
12    friend ostream &operator<<( ostream &, const PhoneNumber & );
13    friend istream &operator>>( istream &, PhoneNumber & );
14 private:
15    string areaCode; // 3-digit area code
16    string exchange; // 3-digit exchange
17    string line; // 4-digit line
18 }; // end class PhoneNumber
19
20 #endif
```

Fig. 11.3 | PhoneNumber class with overloaded stream insertion and stream extraction operators as `friend` functions.

```cpp
// Fig. 11.4: PhoneNumber.cpp
// Overloaded stream insertion and stream extraction operators
// for class PhoneNumber.
#include <iomanip>
#include "PhoneNumber.h"
using namespace std;

// overloaded stream insertion operator; cannot be
// a member function if we would like to invoke it with
// cout << somePhoneNumber;
ostream &operator<<( ostream &output, const PhoneNumber &number )
{
   output << "(" << number.areaCode << ") "
      << number.exchange << "-" << number.line;
   return output; // enables cout << a << b << c;
} // end function operator<<

```

Fig. 11.4 | 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. 11.4 | Overloaded stream insertion and stream extraction operators for class PhoneNumber. (Part 2 of 2.)

```
1  // Fig. 11.5: fig11_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. 11.5 | 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. 11.5 | Overloaded stream insertion and stream extraction operators. (Part 2 of 2.)

- Overloaded operators should mimic the functionality of the built-in counterparts
  - Don't want to overload a “+” to mean subtraction
- Returning a reference from an overloaded >> or << operator is okay, since cout and cin on which they are typically called are global

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

- 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.

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

- 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.
- 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 (don't want to use getters/setters)

11.6 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 in two different ways—either with a parameter that's an object (this requires a copy of the object, so the side effects of the function are *not* applied to the original object), or with a parameter that is a reference to an object (no copy of the original object is made, so all side effects of this function are applied to the original object).

Operator Overloading

- **Operator overloading** lets us define the meaning of an operator when applied to operands of a class type
- Overloaded operators are functions with special names: keyword `operator` followed by the symbol for the operator being defined, e.g.

```
Foo operator+(Foo a, Foo b);
```

- Like all functions, they have a return type, an argument list, and a body
- An overloaded operator has the same number of arguments as operator has operands:
  - a unary operator has one argument
  - a binary operator has two arguments
- No default arguments – except for overloaded function-call operator `()`

Operator Overloading

- For binary operators, left-hand operand is passed to the first argument, the right-hand operand is passed to the second argument
- An overloaded operator function must either be a member of a class or have at least one parameter of class type.
- If an operator function is a member function, the first (left-hand) operand is bound to the implicit `this` pointer.
  - That overloaded operator function than has one less explicit parameter than operator has operands
- Can not define new operators (e.g. can't define ** to mean exponentiation)
- Can not redefine built-in operators for built-in types

```
// error
int operator+(int, int)
```

Operator Overloading

- An overloaded operator has the same precedence and associativity as the corresponding built-in operator.

```
x == y + z;
```

- is always equivalent to

```
x == (y + z)
```

Operators that can be overloaded

- Almost all
  - arithmetic operators: + - * / % and += -= *= /= %= (all binary infix); + - (unary prefix); ++ -- (unary prefix and postfix)
  - bit manipulation: & | ^ << >> and &= |= ^= <<= >>= (all binary infix); ~ (unary prefix)
  - boolean algebra: == != < > <= >= || && (all binary infix); ! (unary prefix)
  - memory management: new new[] delete delete[]
  - implicit conversion operators
  - miscellany: = [] -> , (all binary infix); * & (all unary prefix) () (function call, n-ary infix)

Operator Overloading

- Since overloaded operators are just functions, we can call them directly as well, e.g.

```
data1 + data2;                  // normal expression
operator+(data1, data2); // equivalent function call
```

  - These are equivalent calls: both call the nonmember function `operator+`
- A member operator function can be called explicitly in the same way that we call any other member function:

```
data1 += data2;                  // expression-based "call"
data1.operator+=(data2);  // a member function call
```

  - Equivalent calls to the member function `operator+=`, binding `this` to the address of data1 and passing data2 as an argument.

Which Operators Not to Overload

- Recall that a few operators guarantee the order in which operands are evaluated.
  - e.g. the logical AND (&&) and OR (II), as well as COMMA (,) guarantee that the left operand will be evaluated before the right

    ```
    if (i < n && i++ < n) …
    ```

  - logical AND (&&) and OR (II) also guarantee "short circuit" evaluation, i.e. if the first operand to these operators is sufficient to determine the overall result, evaluation stops and the second operand is not evaluated.

Operators Not to Overload

- Because using an overloaded operator is really a function call, these guarantees do not apply to overloaded operators.
- The users of your class types will expect this behavior and may be unpleasantly surprised when their code breaks

- Also, comma and address-of are defined for class types by default
- Best Practices: Ordinarily, the comma, address-of, logical-and and logical-or should not be overloaded

Operators to Overload

- Once you decide what operations your class will provide, determine if some of the operations logically have the same meaning as a built-in operator – those can be defined as overloaded operator functions, e.g.
  - If the class has an operation to test for equality, define operator==. If == is defined, then operator!= should probably be defined as well
  - If the class has a single, natural ordering operation, define operator<. If the class has operator<, it should probably have all of the relational operators.

Defining Overloaded Operators

- Overloaded operators should behave similarly to the corresponding built-in operators
- The return type of an overloaded operator should correspond to the return from the built-in version
  - The logical and relational operators should return bool,
  - the arithmetic operators should return a value of the class type
  - Assignment and compound assignment should return a reference to the left-hand operand.

Overloading Unary Operators

11.6 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 in two different ways—either with a parameter that's an object (this requires a copy of the object, so the side effects of the function are *not* applied to the original object), or with a parameter that is a constant reference to an object (no copy of the original object is made).

11.7 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.

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

- Suppose that we want to add 1 to the day in an object of class `Date` that represents dates by keeping track of day, month, and year.

```
Date 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 & );
```

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

- The compiler must be able to distinguish between the signatures of the overloaded prefix and postfix increment operator functions.
- This is achieved by adopting a *convention* that postfix operators will have a dummy parameter that would create a function signature different from the prefix operator.
  - When the compiler sees the postincrementing expression d1++, it generates the *member-function call*

```
d1.operator++( 0 )
```

- The prototype for this function for an object of class Date would be:

```
Date operator++( int )
```

11.7 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 );
```

Return Values for Postfix and Prefix Increment

- Prefix increment returns the incremented object, therefore, it can return objects *by reference*, and avoid copying
- Postfix increment operator returns the original value of the object before the increment occurred, and therefore must return a temporary object that contains that value
  - The postfix increment operator must return objects *by value*
- Using postfix increment on objects is therefore less efficient, since it incurs the overhead of creating a temporary object.

11.8 Case Study: A `Date` Class

- Consider a `Date` class, which uses overloaded prefix and postfix increment operators to add 1 to the day in a `Date` object, and changes the month and year when appropriate.

```
1  // Fig. 11.6: Date.h
2  // Date class definition with overloaded increment operators.
3  #ifndef DATE_H
4  #define DATE_H
5
6  #include <iostream>
7  using namespace std;
8
```

Fig. 11.6 | Date class definition with overloaded increment operators. (Part 1 of 2.)

```
 9  class Date
10  {
11     friend ostream &operator<<( 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     const Date &operator+=( 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     int month;
22     int day;
23     int year;
24
25     static const int days[]; // array of days per month
26     void helpIncrement(); // utility function for incrementing date
27  }; // end class Date
28
29  #endif
```

Fig. 11.6 | Date class definition with overloaded increment operators. (Part 2 of 2.)

```
1  // Fig. 11.7: 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 int 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. 11.7 | Date class member- and friend-function definitions.
(Part 1 of 7.)

```
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. 11.7 | Date class member- and friend-function definitions. (Part 2 of 7.)

```
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. 11.7 | Date class member- and `friend`-function definitions. (Part 3 of 7.)

```
58  // add specified number of days to date
59  const Date &Date::operator+=( 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. 11.7 | Date class member- and friend-function definitions.
(Part 4 of 7.)

```
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
```

Fig. 11.7 | Date class member- and friend-function definitions. (Part 5 of 7.)

```
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
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
```

Fig. 11.7 | Date class member- and friend-function definitions. (Part 6 of 7.)

```
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. 11.7 | Date class member- and friend-function definitions.
(Part 7 of 7.)

```
1  // Fig. 11.08: fig11_11.cpp
2  // Date class test program.
3  #include <iostream>
4  #include "Date.h" // Date class definition
5  using namespace std;
6
```

Fig. 11.8 | Date class test program. (Part 1 of 3.)

```
 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;
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
```

Fig. 11.8 | Date class test program. (Part 2 of 3.)

```
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. 11.8 | Date class test program. (Part 3 of 3.)

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

- The `Date` constructor (defined in Fig. 11.7, 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.
- Overloading the prefix increment operator is straightforward.
  - The prefix increment operator (defined in Fig. 11.7, lines 41–45) calls utility function `helpIncrement` (defined in Fig. 11.7, 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 increment the day correctly.

11.8 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.

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

- On entry to postfix `operator++`, we save the current object (`*this`) in `temp` (line 51).
- Next, use `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.

11.10 Case Study: `Array` Class

- Pointer-based arrays have many problems, including:
  - A program can easily "walk off" either end of an array, because *C++ does not check whether subscripts fall outside the range of an array.*
  - Arrays of size *n* must number their elements 0, …, *n* – 1; alternate subscript ranges- are not allowed.
  - An entire array cannot be input or output at once.
  - Two 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 array cannot be assigned to another with the assignment operator.

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

- Consider an example of Array class that provides capabilities similar to `vector`:
  - 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 Arrays with the equality operators == and !=.

```
1  // Fig. 11.9: fig11_09.cpp
2  // Array class test program.
3  #include <iostream>
4  #include "Array.h"
5  using namespace std;
6
7  int main()
8  {
9     Array integers1( 7 ); // seven-element Array
10    Array integers2; // 10-element Array by default
11
12    // print integers1 size and contents
13    cout << "Size of Array integers1 is "
14       << integers1.getSize()
15       << "\nArray after initialization:\n" << integers1;
16
17    // print integers2 size and contents
18    cout << "\nSize of Array integers2 is "
19       << integers2.getSize()
20       << "\nArray after initialization:\n" << integers2;
21
```

Fig. 11.9 | Array class test program. (Part 1 of 7.)

```
22      // input and print integers1 and integers2
23      cout << "\nEnter 17 integers:" << endl;
24      cin >> integers1 >> integers2;
25
26      cout << "\nAfter input, the Arrays contain:\n"
27         << "integers1:\n" << integers1
28         << "integers2:\n" << integers2;
29
30      // use overloaded inequality (!=) operator
31      cout << "\nEvaluating: integers1 != integers2" << endl;
32
33      if ( integers1 != integers2 )
34         cout << "integers1 and integers2 are not equal" << endl;
35
36      // create Array integers3 using integers1 as an
37      // initializer; print size and contents
38      Array integers3( integers1 ); // invokes copy constructor
39
40      cout << "\nSize of Array integers3 is "
41         << integers3.getSize()
42         << "\nArray after initialization:\n" << integers3;
43
```

Fig. 11.9 | Array class test program. (Part 2 of 7.)

```
44      // use overloaded assignment (=) operator
45      cout << "\nAssigning integers2 to integers1:" << endl;
46      integers1 = integers2; // note target Array is smaller
47
48      cout << "integers1:\n" << integers1
49         << "integers2:\n" << integers2;
50
51      // use overloaded equality (==) operator
52      cout << "\nEvaluating: integers1 == integers2" << endl;
53
54      if ( integers1 == integers2 )
55         cout << "integers1 and integers2 are equal" << endl;
56
57      // use overloaded subscript operator to create rvalue
58      cout << "\nintegers1[5] is " << integers1[ 5 ];
59
60      // use overloaded subscript operator to create lvalue
61      cout << "\n\nAssigning 1000 to integers1[5]" << endl;
62      integers1[ 5 ] = 1000;
63      cout << "integers1:\n" << integers1;
64
```

Fig. 11.9 | Array class test program. (Part 3 of 7.)

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

Fig. 11.9 | 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. 11.9 | 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

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

Fig. 11.9 | Array class test program. (Part 6 of 7.)

```
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. 11.9 | Array class test program. (Part 7 of 7.)

11.10 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 linked lists, 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.
  - The STL `map` class redefines the argument to [] to be any keys (e.g. strings), not just integers.

11.10 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<<`, which has similar functionality to our `print_vector<T>` function but are declared as overloaded operators

```
operator<<( cout, arrayObject )
```

- When the compiler sees an expression like `cin >> arrayObject`, it invokes non-member function `operator>>` with the call

```
operator>>( cin, arrayObject )
```

```
1  // Fig. 11.10: Array.h
2  // Array class definition with overloaded operators.
3  #ifndef ARRAY_H
4  #define ARRAY_H
5
6  #include <iostream>
7  using namespace std;
8
9  class Array
10 {
11    friend ostream &operator<<( ostream &, const Array & );
12    friend istream &operator>>( istream &, Array & );
13 public:
14    Array( int = 10 ); // default constructor
15    Array( const Array & ); // copy constructor
16    ~Array(); // destructor
17    int getSize() const; // return size
18
19    const Array &operator=( const Array & ); // assignment operator
20    bool operator==( const Array & ) const; // equality operator
21
```

Fig. 11.10 | 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      int size; // pointer-based array size
35      int *ptr; // pointer to first element of pointer-based array
36   }; // end class Array
37
38   #endif
```

Fig. 11.10 | Array class definition with overloaded operators. (Part 2 of 2.)

Array a;

a[3]

- Returns lvalue

const Array a;

a[3]

- Returns rvalue

```cpp
// Fig 11.11: Array.cpp
// Array class member- and friend-function definitions.
#include <iostream>
#include <iomanip>
#include <cstdlib> // exit function prototype
#include "Array.h" // Array class definition
using namespace std;

// default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
   // validate arraySize
   if ( arraySize > 0 )
      size = arraySize;
   else
      throw invalid_argument( "Array size must be greater than 0" );

   ptr = new int[ size ]; // create space for pointer-based array

   for ( int i = 0; i < size; ++i )
      ptr[ i ] = 0; // set pointer-based array element
} // end Array default constructor
```

Fig. 11.11 | Array class member- and friend-function definitions.
(Part 1 of 8.)

```
23
24 // copy constructor for class Array;
25 // must receive a reference to prevent infinite recursion
26 Array::Array( const Array &arrayToCopy )
27    : size( arrayToCopy.size )
28 {
29    ptr = new int[ size ]; // create space for pointer-based array
30
31    for ( int i = 0; i < size; ++i )
32       ptr[ i ] = arrayToCopy.ptr[ i ]; // copy into object
33 } // end Array copy constructor
34
```

Fig. 11.11 | Array class member- and friend-function definitions. (Part 2 of 8.)

```
35  // destructor for class Array
36  Array::~Array()
37  {
38     delete [] ptr; // release pointer-based array space
39  } // end destructor
40
41  // return number of elements of Array
42  int Array::getSize() const
43  {
44     return size; // number of elements in Array
45  } // end function getSize
46
```

Fig. 11.11 | `Array` class member- and `friend`-function definitions. (Part 3 of 8.)

```
47  // overloaded assignment operator;
48  // const return avoids: ( a1 = a2 ) = a3
49  const Array &Array::operator=( const Array &right )
50  {
51     if ( &right != this ) // avoid self-assignment
52     {
53        // for Arrays of different sizes, deallocate original
54        // left-side array, then allocate new left-side array
55        if ( size != right.size )
56        {
57           delete [] ptr; // release space
58           size = right.size; // resize this object
59           ptr = new int[ size ]; // create space for array copy
60        } // end inner if
61
62        for ( int i = 0; i < size; ++i )
63           ptr[ i ] = right.ptr[ i ]; // copy array into object
64     } // end outer if
65
66     return *this; // enables x = y = z, for example
67  } // end function operator=
68
```

Fig. 11.11 | `Array` class member- and `friend`-function definitions. (Part 4 of 8.)

```
69  // determine if two Arrays are equal and
70  // return true, otherwise return false
71  bool Array::operator==( const Array &right ) const
72  {
73     if ( size != right.size )
74        return false; // arrays of different number of elements
75
76     for ( int i = 0; i < size; ++i )
77        if ( ptr[ i ] != right.ptr[ i ] )
78           return false; // Array contents are not equal
79
80     return true; // Arrays are equal
81  } // end function operator==
82
```

Fig. 11.11 | `Array` class member- and `friend`-function definitions. (Part 5 of 8.)

```
83  // overloaded subscript operator for non-const Arrays;
84  // reference return creates a modifiable lvalue
85  int &Array::operator[]( int subscript )
86  {
87     // check for subscript out-of-range error
88     if ( subscript < 0 || subscript >= size )
89        throw out_of_range( "Subscript out of range" );
90
91     return ptr[ subscript ]; // reference return
92  } // end function operator[]
93
94  // overloaded subscript operator for const Arrays
95  // const reference return creates an rvalue
96  int Array::operator[]( int subscript ) const
97  {
98     // check for subscript out-of-range error
99     if ( subscript < 0 || subscript >= size )
100       throw out_of_range( "Subscript out of range" );
101
102    return ptr[ subscript ]; // returns copy of this element
103 } // end function operator[]
104
```

Fig. 11.11 | `Array` class member- and `friend`-function definitions. (Part 6 of 8.)

```
105 // overloaded input operator for class Array;
106 // inputs values for entire Array
107 istream &operator>>( istream &input, Array &a )
108 {
109    for ( int i = 0; i < a.size; ++i )
110       input >> a.ptr[ i ];
111
112    return input; // enables cin >> x >> y;
113 } // end function
114
```

Fig. 11.11 | `Array` class member- and `friend`-function definitions. (Part 7 of 8.)

```
115 // overloaded output operator for class Array
116 ostream &operator<<( ostream &output, const Array &a )
117 {
118    int i;
119
120    // output private ptr-based array
121    for ( i = 0; i < a.size; ++i )
122    {
123       output << setw( 12 ) << a.ptr[ i ];
124
125       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
126          output << endl;
127    } // end for
128
129    if ( i % 4 != 0 ) // end last line of output
130       output << endl;
131
132    return output; // enables cout << x << y;
133 } // end function operator<<
```

Fig. 11.11 | `Array` class member- and `friend`-function definitions. (Part 8 of 8.)

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

- Line 14 declares the *default constructor* for the class and specifies a default size of 10 elements.
- The default constructor (defined in Fig. 11.11, lines 10–22) 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.

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

- Line 15 declares a *copy constructor* 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.

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

- The copy constructor for `Array` uses a member initializer (line 27) to copy the `size` of the initializer `Array` into data member `size`, uses `new` (line 29) 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) – hence, copy constructors can access argument object's internals

Software Engineering Observation 11.2

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

Common Programming Error 11.3

A copy constructor must receive its argument by reference, not by value. Otherwise, the copy constructor call results in infinite recursion (a fatal logic error) because receiving an object by value requires the copy constructor to make a copy of the argument object. Recall that any time a copy of an object is required, the class's copy constructor is called. If the copy constructor received its argument by value, the copy constructor would call itself recursively to make a copy of its argument!

Common Programming Error 11.4

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 be undefined, 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.

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

- Line 16 declares the class's destructor
- 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 11.2

If after deleting dynamically allocated memory, the pointer will continue to exist in memory, set the pointer's value to 0 to indicate that the pointer no longer points to memory in the free store. By setting the pointer to 0, 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 0, your code could inadvertently access the reallocated memory, causing subtle, nonrepeatable logic errors.

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

- Line 19 declares the overloaded assignment operator function for the class.
- When the compiler sees the expression `integers1 = integers2` in line 46 of Fig. 11.9, the compiler invokes member function `operator=` with the call
  - `integers1.operator=( integers2 )`
- Member function `operator=`'s implementation (Fig. 11.11, lines 49–67) tests for self-assignment (line 51) 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.

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

- `operator=` determines whether the sizes of the two arrays are identical (line 55); in that case, the original array of integers in the left-side `Array` object is not reallocated.
- Otherwise, `operator=` uses `delete` (line 57) to release the memory, copies the `size` of the source array to the `size` of the target array (line 58), uses `new` to allocate memory for the target array and places the pointer returned by `new` into the array's `ptr` member.

- Note that = is right-to-left associative
  - i.e. x = y = z → x = (y = z)
- Regardless of whether this is a self-assignment, `operator=` returns the current object (i.e., *this in line 66) 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 11.3

A copy constructor, a destructor and an overloaded assignment operator are usually provided as a group for any class that uses dynamically allocated memory.

Common Programming Error 11.5

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

Software Engineering Observation 11.4

It's possible to prevent one object of a class from being assigned to another. This is done by declaring the assignment operator as a `private` member of the class.

Software Engineering Observation 11.5

It's possible to prevent class objects from being copied; to do this, simply make both the overloaded assignment operator and the copy constructor of that class `private`.

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

- Line 20 declares the overloaded equality operator (`==`) for the class.
- When the compiler sees the expression `integers1 == integers2` in line 54, the compiler invokes member function `operator==` with the call
  - `integers1.operator==( integers2 )`
- Member function `operator==` immediately returns `false` if the `size` members of the arrays 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.

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

- Lines 23–26 of the header define the overloaded inequality operator (`!=`) for the class.
- Member function `operator!=` uses the overloaded `operator==` function to determine 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.

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

- Lines 29 and 32 declare two overloaded subscript operators
- When the compiler sees the expression `integers1[5]` (line 58), 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[]` when the subscript operator is used on a `const Array` object.
- 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.

11.11 Operator Functions as Class Members vs. Non-Member Functions (cont.)

- 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 saw with **<<** and **>>** the stream insertion and extraction operators).
- 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.

11.11 Operator Functions as Class Members vs. Non-Member Functions (cont.)

- You might choose a non-member function to overload an operator to enable the operator to be *commutative*.
- Consider a class that implements arbitrarily large integers. We would like to enable addition of a fundamental type such as long int and our user-defined type HugeInteger.
- If we define the `operator+` as the member function, we would never be able to invoke it unless `HugeInteger` is on the left.
  - Class object must appear on the left of the overloaded binary operator if it is overloaded as a member function
  - The operator that deals with class objects on the left may be a member function. The non-member function will then just need to swap the arguments and call the member function.

Overloaded function call operator()

- You can overload function call operator () and give it an arbitrary number of parameters
- E.g. for a custom-built String class, you can defined as member function that selects a substring from the string object:

```
String operator()(size_t offset, size_t length);
```

- The operator's two integer parameters specify the start location and the length of the substring to be selected.
- E.g. if str1 is a String object containing the string "AEIOU", when the compiler encounters the expression str1(2, 2), it will generate the member-function call

  str1.operator()(2, 2)

  - which returns a String containing "IO".

Overloaded function call operator()

- You can overload function call operator () and give it an arbitrary number of parameters
- E.g. for a custom-built String class, you can defined as member function that selects a substring from the string object:

```
String operator()(size_t offset, size_t length);
```

- The operator's two integer parameters specify the start location and the length of the substring to be selected.
- E.g. if str1 is a String object containing the string "AEIOU", when the compiler encounters the expression str1(2, 2), it will generate the member-function call

  str1.operator()(2, 2)

  - which returns a String containing "IO".

Functors: Fibonacci Example

```
class Fibonacci {
  int elt1, elt2;
  Int cnt;
  Fibonacci() : elt1(0), elt2(1), cnt(0) {}
  int operator() () {
    cnt++;
    int tmp1 = elt1, tmp2 = elt2;
    elt1 = tmp2;
    elt2 = tmp1 + tmp2;
    return elt2;
  }
}
int main(int argc, char* argv[]) {
  Fibonacci a, b;
  for(int i = 0; i < 20; i++)
    cout << a() << endl;
  cout << a << endl;
  cout << b() << endl;
  cout << b << endl;
}
```