1 Operator Overloading

1.1 Introduction

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

- Use operators with objects (operator overloading)

    - Clearer than function calls for certain classes
    - Operator sensitive to context

- Examples

    - $<<$; Stream insertion, bitwise left-shift
    - $+$; Performs arithmetic on multiple types (integers, floats, etc.)

1.2 Fundamentals of Operator Overloading

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

- Types

    - Built in (**int**, **char**) or user-defined
    - Can use existing operators with user-defined types; Cannot create new operators

- Overloading operators

    - Create a function for the class
    - Name function **operator** followed by symbol; **Operator+** for the addition operator $+$

- Using operators on a class object

    - It must be overloaded for that class
        * Exceptions:
        * Assignment operator, $=$; Memberwise assignment between objects
        * Address operator, **&**; Returns address of object

∗ Both can be overloaded

- Overloading provides concise notation

  - **object2 = object1.add(object2);**
  - **object2 = object2 + object1;**

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

1.3   Restrictions on Operator Overloading

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

- Cannot change

  - How operators act on built-in data types; i.e., cannot change integer addition
  - Precedence of operator (order of evaluation); Use parentheses to force order-of-operations
  - Associativity (left-to-right or right-to-left)
  - Number of operands; **&** is unitary, only acts on one operand

- Cannot create new operators

- Operators must be overloaded explicitly; Overloading **+** does not overload **+=**

1.4   Operator Functions As Class Members Vs.   As Friend Functions

- Operator functions

  - Member functions

    ∗ Use **this** keyword to implicitly get argument

* Gets left operand for binary operators (like +)
        * Leftmost object must be of same class as operator
    – Non member functions
        * Need parameters for both operands
        * Can have object of different class than operator
        * Must be a **friend** to access **private** or **protected** data
    – Called when
        * Left operand of binary operator of same class
        * Single operand of unitary operator of same class

- Overloaded << operator

    – Left operand of type **ostream &**; Such as **cout** object in **cout << classObject**
    – Similarly, overloaded >> needs **istream &**
    – Thus, both must be non-member functions

- Commutative operators

    – May want + to be commutative; So both "**a + b**" and "**b + a**" work
    – Suppose we have two different classes
    – Overloaded operator can only be member function when its class is on left
        * **HugeIntClass + Long int**
        * Can be member function
    – When other way, need a non-member overload function; **Long int + HugeIntClass**

1.5 Overloading Stream-Insertion and Stream-Extraction Operators

- << and >>

    – Already overloaded to process each built-in type
    – Can also process a user-defined class

- Example program

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

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

```
1   // Fig. 8.3: fig08_03.cpp
2   // Overloading the stream-insertion and
3   // stream-extraction operators.
4   #include <iostream>
5
6   using std::cout;
7   using std::cin;
8   using std::endl;
9   using std::ostream;
10  using std::istream;
11
12  #include <iomanip>
13
14  using std::setw;
15
16  // PhoneNumber class definitio
17  class PhoneNumber {
18     friend ostream &operator<<(
19     friend istream &operator>>(
20
21  private:
22     char areaCode[ 4 ];   // 3-d
23     char exchange[ 4 ];   // 3-digit exchange and null
24     char line[ 5 ];       // 4-digit line and null
25
26  }; // end class PhoneNumber
```

Outline

fig08_03.cpp
(1 of 3)

Notice function prototypes for overloaded operators **>>** and **<<**

They must be non-member `friend` functions, since the object of class `Phonenumber` appears on the right of the operator.

`cin << object`
`cout >> object`

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

```
27
28    // overloaded stream-insertion operator; cannot be
29    // a member function if we would like to invoke it with
30    // cout << somePhoneNumber;
31    ostream &operator<<( ostream &output, const PhoneNumber &num )
32    {
33        output << "(" << num.areaCode << ") "
34               << num.exchange << "-" << num.line;
35
36        return output;        // enables cout << a << b << c;
37
38    } // end function operator<<
39
40    // overloaded stream-extraction operator; cannot be
41    // a member function if we
42    // cin >> somePhoneNumber;
43    istream &operator>>( istream
44    {
45        input.ignore();                    // skip (
46        input >> setw( 4 ) >> num.areaCode; // input are
47        input.ignore( 2 );
48        input >> setw( 4 ) >> num.exchange;
49        input.ignore();
50        input >> setw( 5 ) >> num.line;
51
52        return input;        // enables cin >
```

The expression:
cout << phone;
is interpreted as the function call:
operator<<(cout, phone);

output is an alias for **cout**.

This allows objects to be cascaded.
<< phone1 << phone2;
alls
ator<<(cout, phone1), and
returns **cout**.

Next, **cout << phone2** executes.

ignore() skips specified number of characters from input (1 by default).

Stream manipulator **setw** restricts number of characters read. **setw(4)** allows 3 characters to be read, leaving room for the null character.

```
53
54    } // end function operator>>
55
56    int main()
57    {
58        PhoneNumber phone; // create object phone
59
60        cout << "Enter phone number in the form (123) 456-7890:\n";
61
62        // cin >> phone invokes operator>> by implicitly issuing
63        // the non-member function call operator>>( cin, phone )
64        cin >> phone;
65
66        cout << "The phone number entered was: " ;
67
68        // cout << phone invokes operator<< by implicitly issuing
69        // the non-member function call operator<<( cout, phone )
70        cout << phone << endl;
71
72        return 0;
73
74    } // end main
```

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

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

1.6 Overloading Unary Operators

- Overloading unary operators

  - Non-**static** member function, no arguments
  - Non-member function, one argument; Argument must be class object or reference to class object
  - Remember, **static** functions only access **static** data

- Upcoming example (8.10)

  - Overload **!** to test for empty string
  - If non-**static** member function, needs no arguments
    * **!s** becomes **s.operator!()**
    * **class String { public: bool operator!() const; . . . };**

- If non-member function, needs one argument

  - **s!** becomes **operator!(s)**
  - **class String { friend bool operator!(const String &) ... }**

1.7 Overloading Binary Operators

- Overloading binary operators

  - Non-**static** member function, one argument
  - Non-member function, two arguments
  - One argument must be class object or reference

- Upcoming example

  - If non-**static** member function, needs one argument
    * **class String {**
    * **public:**
    * **const String &operator+=(const String &);**
    * **...**
    * **};**
  - **y += z** equivalent to **y.operator+=(z)**

6

1.8   Case Study: Array class

- Arrays in C++

  - No range checking
  - Cannot be compared meaningfully with ==
  - No array assignment (array names **const** pointers)
  - Cannot input/output entire arrays at once; One element at a time

- Example:Implement an Array class with

  - Range checking
  - Array assignment
  - Arrays that know their size
  - Outputting/inputting entire arrays with << and >>
  - Array comparisons with == and !=

- Copy constructor

  - Used whenever copy of object needed
    * Passing by value (return value or parameter)
    * Initializing an object with a copy of another; **Array newArray(oldArray);**
    * **newArray** copy of **oldArray**
  - Prototype for class **Array**
    * **Array(const Array &);**
    * *Must* take reference
      · Otherwise, pass by value
      · Tries to make copy by calling copy constructor . . .
      · Infinite loop

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

```
1   // Fig. 8.4: array1.h
2   // Array class for storing arrays of integers.
3   #ifndef ARRAY1_H
4   #define ARRAY1_H
5
6   #include <iostream>
7
8   using std::ostream;
9   using std::istream;
10
11  class Array {
12     friend ostream &operator<<( ostream &, const Array & );
13     friend istream &operator>>( istream &, Array & );
14
15  public:
16     Array( int = 10 );        // default cons
17     Array( const Array & );   // copy constru
18     ~Array();                 // destructor
19     int getSize() const;      // return s
20
21     // assignment operator
22     const Array &operator=( const Array & );
23
24     // equality operator
25     bool operator==( const Array & ) const;
26
```

Most operators overloaded as member functions (except << and >>, which must be non-member functions).

Prototype for copy constructor.

```
27     // inequality operator; returns opposite of == operator
28     bool operator!=( const Array &right ) const
29     {
30        return ! ( *this == right ); // invokes Array::operator==
31
32     } // end function operator!=
33
34     // subscript operator for non-con
35     int &operator[]( int );
36
37     // subscript operator for const objects returns rvalue
38     const int &operator[]( int ) const;
39
40  private:
41     int size; // array size
42     int *ptr; // pointer to first element of array
43
44  }; // end class Array
45
46  #endif
```

!= operator simply returns opposite of == operator. Thus, only need to define the == operator.

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

8

```
1   // Fig 8.5: array1.cpp
2   // Member function definitions for class Array
3   #include <iostream>
4
5   using std::cout;
6   using std::cin;
7   using std::endl;
8
9   #include <iomanip>
10
11  using std::setw;
12
13  #include <new>        // C++ standard "new" operator
14
15  #include <cstdlib>    // exit function prototype
16
17  #include "array1.h"  // Array class definition
18
19  // default constructor for class Array (default size 10)
20  Array::Array( int arraySize )
21  {
22     // validate arraySize
23     size = ( arraySize > 0 ? arraySize : 10 );
24
25     ptr = new int[ size ]; // create space for array
26
```

```
27     for ( int i = 0; i < size; i++ )
28        ptr[ i ] = 0;           // initialize array
29
30  } // end Array default constructor
31
32  // copy constructor for class Array;
33  // must receive a reference to pre
34  Array::Array( const Array &arrayTo
35     : size( arrayToCopy.size )
36  {
37     ptr = new int[ size ]; // create space for array
38
39     for ( int i = 0; i < size; i++ )
40        ptr[ i ] = arrayToCopy.ptr[ i ];  // copy into object
41
42  } // end Array copy constructor
43
44  // destructor for class Array
45  Array::~Array()
46  {
47     delete [] ptr;  // reclaim array space
48
49  } // end destructor
50
```

We must declare a new integer array so the objects do not point to the same memory.

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

9

```
51  // return size of array
52  int Array::getSize() const
53  {
54     return size;
55
56  } // end function getSize
57
58  // overloaded assignment operator
59  // const return avoids: ( a1 = a2 ) = a3
60  const Array &Array::operator=( const Array &right )
61  {
62     if ( &right != this ) {  // check for self-assignment
63
64        // for arrays of different sizes, deallocate original
65        // left-side array, then allocate new left-side array
66        if ( size != right.size ) {
67           delete [] ptr;       // reclaim space
68           size = right.size;    // resize this object
69           ptr = new int[ size ]; // create space for array copy
70
71        } // end inner if
72
73        for ( int i = 0; i < size; i++ )
74           ptr[ i ] = right.ptr[ i ];  // copy array into object
75
76     } // end outer if
```

Want to avoid self-assignment.

```
77
78     return *this;   // enables x = y = z, for example
79
80  } // end function operator=
81
82  // determine if two arrays are equal and
83  // return true, otherwise return false
84  bool Array::operator==( const Array &right ) const
85  {
86     if ( size != right.size )
87        return false;    // arrays of different sizes
88
89     for ( int i = 0; i < size; i++ )
90
91        if ( ptr[ i ] != right.ptr[ i ] )
92           return false; // arrays are not equal
93
94     return true;        // arrays are equal
95
96  } // end function operator==
97
```

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

```
98   // overloaded subscript operator for non-const Arrays
99   // reference return creates an lvalue
100  int &Array::operator[]( int subscript )
101  {
102     // check for subscript out of range erro
103     if ( subscript < 0 || subscript >= size
104        cout << "\nError: Subscript " << subs
105           << " out of range" << endl;
106
107        exit( 1 );  // terminate program; subscript out of range
108
109     } // end if
110
111     return ptr[ subscript ]; // reference return
112
113  } // end function operator[]
114
```

integers1[5] calls
integers1.operator[](5)

exit() (header <cstdlib>) ends
the program.

```
115  // overloaded subscript operator for const Arrays
116  // const reference return creates an rvalue
117  const int &Array::operator[]( int subscript ) const
118  {
119     // check for subscript out of range error
120     if ( subscript < 0 || subscript >= size ) {
121        cout << "\nError: Subscript " << subscript
122           << " out of range" << endl;
123
124        exit( 1 );  // terminate program; subscript out of range
125
126     } // end if
127
128     return ptr[ subscript ]; // const reference return
129
130  } // end function operator[]
131
132  // overloaded input operator for class Array;
133  // inputs values for entire array
134  istream &operator>>( istream &input, Array &a )
135  {
136     for ( int i = 0; i < a.size; i++ )
137        input >> a.ptr[ i ];
138
139     return input;   // enables cin >> x >> y;
140
141  } // end function
```

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

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

Outline

array1.cpp (7 of 7)

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

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

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

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

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

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

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

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

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

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

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

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

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

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13
```

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

15

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

Attempt to assign 1000 to integers1[15]

Error: Subscript 15 out of range
```

fig08_06.cpp
output (3 of 3)

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