W8.2 Operator Overloading

- Fundamentals of Operator Overloading
- Restrictions on Operator Overloading
- Operator Functions as Class Members vs. as friend Functions
- Overloading Stream Insertion and Extraction Operators
- Overloading Unary Operators
- Overloading Binary Operators
- Case Study: An Array Class
- Converting between Types
- Case Study: A String Class
- Overloading ++ and --
- Case Study: A Date Class

Introduction

- Operator overloading
  - Enabling C++’s operators to work with class objects
  - Using traditional operators with user-defined objects
  - Requires great care, when overloading is misused, program difficult to understand
  - Examples of already overloaded operators
    - Operator << is both the stream-insertion operator and the bitwise left-shift operator
    - + and - perform arithmetic on multiple types
  - Compiler generates the appropriate code based on the manner in which the operator is used

Fundamentals of Operator Overloading

- Overloading an operator
  - Write function definition as normal
  - Function name is keyword `operator` followed by the symbol for the operator being overloaded
  - `operator+` used to overload the addition operator (`+`)
- Using operators
  - To use an operator on a class object it must be overloaded unless the assignment operator (=) or the address operator (&)
  - Assignment operator by default performs memberwise assignment
  - Address operator (&) by default returns the address of an object

Restrictions on Operator Overloading

- C++ operators that can be overloaded

<table>
<thead>
<tr>
<th>Operators that can be overloaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
</tr>
<tr>
<td>+=</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
</tr>
</tbody>
</table>

- C++ Operators that cannot be overloaded

<table>
<thead>
<tr>
<th>Operators that cannot be overloaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
</tr>
</tbody>
</table>

Operator Functions as Class Members vs. as friend Functions

- Member vs non-member
  - Operator functions can be member or non-member functions
  - When overloading (.), [ ], -> or any of the assignment operators, must use a member function
- Operator functions as member functions
  - Leftmost operand must be an object (or reference to an object) of the class
  - If left operand is a different type, operator function must be a non-member function
- Operator functions as non-member functions
  - Must be friends if needs to access private or protected members
  - Enable the operator to be commutative (a*b, b*a)
Overloading Stream-Insertion and Stream-Extraction Operators

- Overloaded << and >> operators
  - Overloaded to perform input/output for user-defined types
  - Left operand of types ostream & and istream &
  - Must be a non-member function because left operand is not an object of the class
  - Must be a friend function to access private data members

Outline

1. Class definition
   1.1 Function definition
   1.2 Initialize variables
2. Get input
   2.1 Assign to object
   2.2 Output data

Program Output

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

Overloading Unary Operators

- Overloading unary operators
  - Can be overloaded with no arguments or one argument
  - Should usually be implemented as member functions
    - Avoid friend functions and classes because they violate the encapsulation of a class
  - Example declaration as a member function:
    ```cpp
    class String {
    public:
    friend bool operator!( const String & ) const;
    ...
    }
    ```

- Example declaration as a non-member function
  ```cpp
  class String {...
  friend bool operator!( const String & )
  ...
  }
  ```
Overloading Binary Operators

- Overloaded Binary operators
  - Non-static member function, one argument
    Example:
    ```
    class String {
    public:
        const String &operator+=( const String & );
    ... 
    };
    - y += z is equivalent to y.operator+=( z )
    ```
254   integers1[ 15 ] = 1000;  // ERROR: out of range
253   cout << "Attempt to assign 1000 to integers1[15]" << endl;
248   integers1[ 5 ] = 1000;
247   cout << "Assigning 1000 to integers1[5]
";  
';
242
239   cout << "Evaluating: integers1 == integers2
";
236        << "integers2:
" << integers2 << '
';
234   integers1 = integers2;
232   // use overloaded assignment (=) operator
230        << integers3 << '
';
229        << "
Array after initialization:
";
227
191   cout << "Size of array integers3 is 7
";
189{
187   int main()
186   {
185   #include "array1.h"
183   using std::endl;
182   using std::cin;
180   #include <iostream>
179   // Driver for simple class Array
178   // Fig. 8.4: fig08_04.cpp
177}
173   if ( i % 4 != 0 )
172
171   }
168
166   for ( i = 0; i < a.size; i++ ) {
164   assert( 0 <= subscript && subscript < size );
162ostream &operator<<( ostream &output, const Array &a )
160   output << "\n";
159}  
158   return input;   // enables cin >> x >> y;
155   for ( int i = 0; i < a.size; i++ )
154{
152// inputs values for entire array.
151// Overloaded input operator for class Array;
150
148// static functions cannot be const
146   return ptr[ subscript ]; // const reference return
143
140{
137// Overloaded subscript operator for const Arrays
136
135}
133
125   return true;        // arrays are equal
124         return false; // arrays are not equal
120   for ( int i = 0; i < size; i++ )
119      return false;    // arrays of different sizes
116{
115bool Array::operator==( const Array &right ) const
112   return input;   // enables cin >> x >> y;
110
109   }  
106      for ( int i = 0; i < size; i++ )
105         ptr = new int[ size ]; // create space for array copy
104         size = right.size;     // resize this object
101         size = right.size;     // resize this object
100   return true;   // arrays are equal
97   cout << "They are not equal
";
95   cout << "Evaluating: integers1 != integers2
";
94   // use overloaded inequality (!=) operator
93
91   cout << "Array after initialization:
";
90        << integers2.getSize() << '
';
89        << Array::getArrayCount() << "\n\n";
87        << "\nArray after initialization:
";
86        << integers2 << '
';
85        << "
Array after initialization:
";
83        << integers2 << '
';
82        << "
Array after initialization:
";
81        << integers2 << '
';
80        << "
Array after initialization:
";
79        << integers2 << '
';
78        << "
Array after initialization:
";
77        << integers2 << '
';
76        << "
Array after initialization:
";
75        << integers2 << '
';
74        << "
Array after initialization:
";
73        << integers2 << '
';
72        << "
Array after initialization:
";
71        << integers2 << '
';
70        << "
Array after initialization:
";
69        << integers2 << '
';
68        << "
Array after initialization:
";
67        << integers2 << '
';
66        << "
Array after initialization:
";
65        << integers2 << '
';
64        << "
Array after initialization:
";
63        << integers2 << '
';
62        << "
Array after initialization:
";
61        << integers2 << '
';
60        << "
Array after initialization:
";
59        << integers2 << '
';
58        << "
Array after initialization:
";
57        << integers2 << '
';
56        << "
Array after initialization:
";
55        << integers2 << '
';
54        << "
Array after initialization:
";
53        << integers2 << '
';
52        << "
Array after initialization:
";
51        << integers2 << '
';
50        << "
Array after initialization:
";
49        << integers2 << '
';
48        << "
Array after initialization:
";
47        << integers2 << '
';
46        << "
Array after initialization:
";
45        << integers2 << '
';
44        << "
Array after initialization:
";
43        << integers2 << '
';
42        << "
Array after initialization:
";
41        << integers2 << '
';
40        << "
Array after initialization:
";
39        << integers2 << '
';
38        << "
Array after initialization:
";
37        << integers2 << '
';
36        << "
Array after initialization:
";
35        << integers2 << '
';
34        << "
Array after initialization:
";
33        << integers2 << '
';
32        << "
Array after initialization:
";
31        << integers2 << '
';
30        << "
Array after initialization:
";
29        << integers2 << '
';
28        << "
Array after initialization:
";
27        << integers2 << '
';
26        << "
Array after initialization:
";
25        << integers2 << '
';
24        << "
Array after initialization:
";
23        << integers2 << '
';
22        << "
Array after initialization:
";
21        << integers2 << '
';
20        << "
Array after initialization:
";
19        << integers2 << '
';
18        << "
Array after initialization:
";
17        << integers2 << '
';
16        << "
Array after initialization:
";
15        << integers2 << '
';
14        << "
Array after initialization:
";
13        << integers2 << '
';
12        << "
Array after initialization:
";
11        << integers2 << '
';
10        << "
Array after initialization:
";
9         << integers2 << '
';
8         << "
Array after initialization:
";
7         << integers2 << '
';
6         << "
Array after initialization:
";
5         << integers2 << '
';
4         << "
Array after initialization:
";
3         << integers2 << '
';
2         << "
Array after initialization:
";
1         << integers2 << '
';
0         << "
Array after initialization:
";

// inputs values for entire array.
// Overloaded input operator for class Array;
// static functions cannot be const
    return ptr[ subscript ]; // const reference return

// Overloaded subscript operator for const Arrays
}
Program Output

Evaluating: integers1 == integers2
They 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]
Assertion failed: 0 <= subscript && subscript < size, file Array1.cpp, line 95 abnormal program termination

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

Converting between Types

- Cast operator
  - Forces conversions among built-in types
  - Specifies conversions between user-defined and built-in types
  - Conversion operator must be a non-static member function
  - Cannot be a friend function
  - Do not specify return type

  - Return type is the type to which the object is being converted

- For user-defined class A
  - A::operator char *() const;
    - Declares an overloaded cast operator function for creating a char * out of an A object

Case Study: A String Class

- Build a class to handle strings
  - Class string in standard library (more Chapter 19)

- Conversion constructor
  - Single-argument constructors that turn objects of other types into class objects

- Compiler and casting
  - Casting can prevent the need for overloading
  - If an object s of user-defined class String appears in a program where an ordinary char * is expected, such as
    cout << s;
The compiler calls the overloaded cast operator function operator char * to convert the object into a char * and uses the resulting char * in the expression

```
// Fig. 8.5: string1.h
// Definition of a String class
#ifndef STRING1_H
#define STRING1_H

#include <iostream>

using std::ostream;
using std::istream;

class String {
  friend ostream &operator<<( ostream &, const String & );
  friend istream &operator>>( istream &, String & );

public:
  String( const char * = "" ); // conversion/default ctor
  String( const String & );    // copy constructor
  ~String();                   // destructor
  const String &operator=( const String & );  // assignment
  const String &operator+=( const String & ); // concatenation
  bool operator!() const;                  // is String empty?
  bool operator==( const String & ) const; // test s1 == s2
  bool operator<( const String & ) const;  // test s1 < s2
  bool operator!=( const String & right ) const
    { return !( *this == right ); }
  bool operator>( const String &right ) const
    { return right < *this; }
  // test s1 != s2
  bool operator<=( const String &right ) const
    { return !( right < *this ); }
  // test s1 >= s2
  bool operator>=( const String &right ) const
    { return !( *this < right ); }
  // test s1 > s2
  char &operator[]( int );             // subscript operator
  const char &operator[]( int ) const; // subscript operator
  String operator[]( int, int );       // return a substring
  int getLength() const;               // return string length

private:
  int length;                   // string length
  char *sPtr;                   // pointer to start of string

  void setString( const char * );  // utility function
};
#endif

// Fig. 8.5: string1.cpp
// Member function definitions for class String
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::setw;
```
```cpp
#include "string1.h"

using std::cout;

// Driver for class String

    input >> setw(100) >> temp;
    char temp[100];   // buffer to store input

    istream &operator>>( istream &input, String &s )
        // Overloaded input operator

    
        // Overloaded output operator

        int len;
        // determine length of substring
        // ensure index is in range and substring length >= 0

        return sPtr[ subscript ];  // creates rvalue

        // First test for subscript out of range

        const char &String::operator[] ( int subscript ) const
            // Return a reference to a character in a String as an rvalue.

        char &String::operator[] ( int subscript )
            // Function call operator.

        const String &String::operator= ( const String &right )
            // Overloaded = operator; avoids self assignment

        Constructors and destructors

        Destructor:  to you
        s1 += s2 yields s1 = happy birthday
        Testing !s3:
        s2 <= s1 yields true
        s2 >= s1 yields false
        s2 <  s1 yields true
        s2 != s1 yields true
        s2 == s1 yields false
        The results of comparing s2 and s1:
        s1 is "happy birthday"; s2 is "happy birthday to you"
```
Case Study: A Date Class

- The following example creates a Date class with
  - An overloaded increment operator to change the day, month and year
  - An overloaded == operator
  - A function to test for leap years
  - A function to determine if a day is last day of a month

1.2 Member variables

- Added data members to return the day of the week and month

2. Function calls

- Function calls to demonstrate the functionality of the Date class

Program Output

- Output of the program calls to the Date class functions

Overloading ++ and --

- Pre/post incrementing/decrementing operators
  - Allowed to be overloaded
  - Distinguishing between pre and post operators
    - Prefix versions are overloaded the same as other prefix unary operators
      - `d1.operator++();`  // for ++d1
        - Convention adopted that when compiler sees postincrementing expression, it will generate the member-function call
      - `d1.operator++( 0 );`  // for d1++
        - 0 is a dummy value to make the argument list of `operator++` distinguishable from the argument list of `operator++(0)`
Testing the postincrement operator:

```
d4 is March 19, 1969
++d4 is March 19, 1969
d4 is March 18, 1969
```

Testing the preincrement operator:

```
++d3 is February 29, 1992
d3 is February 28, 1992
d3 is January 1, 1900
```

```cpp
#include <iostream>

using std::cout;

#include "date1.h"

Date d1, d2( 12, 27, 1992 ), d3( 0, 99, 8045 );

int main()
{
    cout << "d1 is " << d1 << endl;
    cout << "d2 += 7 is " << ( d2 += 7 ) << endl;
}
```

```cpp
// Fig. 8.6: fig08_06.cpp
// Driver for class Date
// Fig. 8.6: fig08_06.cpp

const int Date::days[] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };  // days in each month

Date Date::operator+( int additionalDays )
// Add a specific number of days to a date
{
    const Date temp = *this;
    helpIncrement();
    for ( int i = 0; i < additionalDays; i++ )
    {
        helpIncrement();
    }
    return temp;  // value return; not a reference return
}

Date Date::operator++( int )
// parameter name.
// Note that the dummy integer parameter does not have a
// value.
{
    using std::cout;
    using std::endl;

    int day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
    int month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
    int year = ( yy >= 1900 ) ? yy : 1900;

    helpIncrement();
    return *this;  // reference return to create an lvalue
}
```