Operator Overloading

Lab 8: Huge Integer

Rung-Bin Lin

International Program in Informatics for Bachelor
Yuan Ze University

04/15/2021

Objectives

- **Discuss more advanced topics about operator overloading**
  - **Overloading as global functions**
  - **Overloading ++ and –**

Review of Operator Overloading

- Operators that are overloaded as non-static member functions
  - The **leftmost operand** must be an object of the operator's class.
  - Like addition operator X + Y if X is an object of the operator's class, but Y may be an object of the operator's class.
- Operators that are overloaded as global functions
  - The leftmost operand may be an object of a different type or a fundamental type. Like <<, >>, …operators.
  - Usually make friend to the class whose objects will use the operator.
- Operator precedence can not be changed by overloading
- No new operators can be created.

Operators Overloaded as Global Functions

```
class Array {
   friend ostream &operator<<( ostream &, const Array & );
   friend istream &operator>>( istream &, Array & );
public:
   Array( int = 10 ); // default constructor
   …
}

istream &operator>>( istream &input, Array &a )
{
   for ( int i = 0; i < a.size; i++ )
      input >> a.ptr[ i ];

   return input; // enables cin >> x >> y;
} // end function
```

Usage: cin >> A;

Compiler translates this statement into a function call: **Operator>>(cin, A);**

Overloading ++

```
// Figure 11.9
class Date
{
   friend ostream &operator<<( ostream &, const Date & );
public:
   Date( int m = 1, int d = 1, int y = 1900 ); // default constructor
   void setDate( int, int, int ); // set month, day, year
   Date &operator++(); // prefix increment operator
   Date operator++( int ); // postfix increment operator
   const Date &operator+=( int ); // add days, modify object
   static bool leapYear( int ); // is date in a leap year?
   bool endOfMonth( int ) const; // is date at the end of month?
private:
   int month;
   int day;
   int year;

   static const int days[]; // array of days per month
   void helpIncrement(); // utility function for incrementing date
}; // end class Date
```

Overloading Prefix Increment Operator

```
Date &Date::operator++()
{
   helpIncrement(); // increment date
   return *this; // reference return to create an lvalue
} // end function operator++
```

Overloading Postfix Increment Operator

```
// overloaded postfix increment operator; note that the
// dummy integer parameter does not have a parameter name
Date Date::operator++( int )
{
   Date temp = *this; // hold current state of object
   helpIncrement();

   // return unincremented, saved, temporary object
   return temp; // value return; not a reference return
} // end function operator++
```

Use of ++

```
int main()
{
   Date d4( 7, 13, 2002 );

   cout << "\n\nTesting the prefix ++ :\n“ << "  d4 is " << d4 << endl;
   cout << "++d4 is " << ++d4 << endl;
   cout << "  d4 is " << d4;

   cout << "\n\nTesting the postfix ++:\n“ << "  d4 is " << d4 << endl;
   cout << "d4++ is " << d4++ << endl;
   cout << "  d4 is " << d4 << endl;
} // end main
```

Lab 8: Operator Overloading- HugeInt class

- **Take the code for class HugeInt in Fig. 11.22 ~ Fig. 11.24 and create or overload the following operators that makes main() function work correctly. Assume all HugeInts are greater than or equal to zero.**
  - Overload **operator + to perform int + HugeInt. This may require overloading the operator as a global function.**
  - Overload the **operator ++ to perform prefix and postfix increment**. You can refer to example in Fig. 11.10.
  - Overload the **operator >= to perform a comparison of HugeInt >= HugeInt, HugeInt >= int, int >= HugeInt, HugeInt >= string, and string >= HugeInt, where string contains only decimal digits. If the comparison is true, return true, otherwise, return false.**
- **The main() function is given and should not be modified.**

Main() Function

```
int main()
{
   HugeInt n1( 7654321 );
   HugeInt n3( "99999999999999999999999999999" );
   HugeInt n4( "1" );
   HugeInt n5(n4);

   cout << "n1 is " << n1 << "\nn3 is " << n3
      << "\nn4 is " << n4 << "\nn5 is " << n5 << "\n\n";
   HugeInt n6 = n3 + n4;
   cout <<"n6 = " << n3 << " + " << n4 << " = " << n6 << "\n\n";
   cout << "9 + n1 = " << 9 + n1 << "   " << "9" + n1 << "   " << n1+9 << endl;
   cout << "n4+100+900+n5= " << n4+100+"900"+n5 << endl;

   cout << "n3++ = " << n3++ << endl;
   cout << "n3 = " << n3 << endl;
   cout << "++n3 = " << ++n3 << endl;
   cout << "n3 = " << n3 <<endl;
   if(n3 >= n1)
    cout << "\nyes-1" << endl;
   else cout << "\nno-1" << endl;
    if(n3 >= 100)
    cout << "yes-2" << endl;
   else cout << "no-2" << endl;
    if(100 >= n3)
    cout << "yes-3" << endl;
   else cout << "no-3" << endl;
   if(n3 >= "100")
    cout << "yes-4" << endl;
   else cout << "no-4" << endl;
    if("100" >= n3)
    cout << "yes-5" << endl;
   else cout << "no-5" << endl;
} // end main
```

Key Points for Grading

- All the outputs should be correct. That is, they should be exactly the same as those in the example output.
- The main() function should not be modified.

Output

```
n1 is 7654321
n3 is 99999999999999999999999999
n4 is 1
n5 is 1

n6 = 99999999999999999999999999 + 1 = 100000000000000000000000000

9 + n1 = 7654330    7654330    7654330
n4+100+900+n5= 1002
n3++ = 99999999999999999999999999
n3 = 10000000000000000000000000
++n3 = 10000000000000000000000001
n3 = 10000000000000000000000001

yes-1
yes-2
no-3
yes-4
no-5
```

These five lines must be in “yes-yes-no-yes-no” sequence.

```
// Fig. 11.23: Hugeint.h
// HugeInt class definition.
#ifndef HUGEINT_H
#define HUGEINT_H
#include <iostream>
#include <string>
using namespace std;

class HugeInt
{
   friend ostream &operator<<( ostream &, const HugeInt & );
public:
   static const int digits = 30; // maximum digits in a HugeInt
   HugeInt( long = 0 ); // conversion/default constructor
   HugeInt( const string & ); // conversion constructor
   // addition operator; HugeInt + HugeInt
   HugeInt operator+( const HugeInt & ) const;
   // addition operator; HugeInt + int
   HugeInt operator+( int ) const;
   // addition operator;
   // HugeInt + string that represents large integer value
   HugeInt operator+( const string & ) const;
private:
   short integer[ digits ];
}; // end class HugetInt
#endif
```

```
// Fig. 11.24: Hugeint.cpp
// HugeInt member-function and friend-function definitions.
#include <cctype> // isdigit function prototype
#include "Hugeint.h" // HugeInt class definition
using namespace std;

// default constructor; conversion constructor that converts
// a long integer into a HugeInt object
HugeInt::HugeInt( long value )
{
   // initialize array to zero
   for ( int i = 0; i < digits; i++ )
      integer[ i ] = 0;

   // place digits of argument into array
   for ( int j = digits - 1; value != 0 && j >= 0; j-- )
   {
      integer[ j ] = value % 10;
      value /= 10;
   } // end for
} // end HugeInt default/conversion constructor

// conversion constructor that converts a character string
// representing a large integer into a HugeInt object
HugeInt::HugeInt( const string &number )
{
   // initialize array to zero
   for ( int i = 0; i < digits; i++ )
      integer[ i ] = 0;

   // place digits of argument into array
   int length = number.size();

   for ( int j = digits - length, k = 0; j < digits; j++, k++ )
      if ( isdigit( number[ k ] ) ) // ensure that character is a digit
         integer[ j ] = number[ k ] - '0';
} // end HugeInt conversion constructor
```

```
// conversion constructor that converts a character string
// representing a large integer into a HugeInt object
HugeInt::HugeInt( const string &number )
{
   // initialize array to zero
   for ( int i = 0; i < digits; i++ )
      integer[ i ] = 0;

   // place digits of argument into array
   int length = number.size();

   for ( int j = digits - length, k = 0; j < digits; j++, k++ )
      if ( isdigit( number[ k ] ) ) // ensure that character is a digit
         integer[ j ] = number[ k ] - '0';
} // end HugeInt conversion constructor
```

```
// addition operator; HugeInt + HugeInt
HugeInt HugeInt::operator+( const HugeInt &op2 ) const
{
   HugeInt temp; // temporary result
   int carry = 0;

   for ( int i = digits - 1; i >= 0; i-- )
   {
      temp.integer[ i ] = integer[ i ] + op2.integer[ i ] + carry;

      // determine whether to carry a 1
      if ( temp.integer[ i ] > 9 )
      {
         temp.integer[ i ] %= 10;  // reduce to 0-9
         carry = 1;
      } // end if
      else // no carry
         carry = 0;
   } // end for
```

```
// addition operator; HugeInt + int
HugeInt HugeInt::operator+( int op2 ) const
{
   // convert op2 to a HugeInt, then invoke
   // operator+ for two HugeInt objects
   return *this + HugeInt( op2 );
} // end function operator+

// addition operator;
// HugeInt + string that represents large integer value
HugeInt HugeInt::operator+( const string &op2 ) const
{
   // convert op2 to a HugeInt, then invoke
   // operator+ for two HugeInt objects
   return *this + HugeInt( op2 );
} // end operator+
```

```
// overloaded output operator
ostream& operator<<( ostream &output, const HugeInt &num )
{
   int i;

   for ( i = 0; ( num.integer[ i ] == 0 ) && ( i <= HugeInt::digits ); i++ )
      ; // skip leading zeros

   if ( i == HugeInt::digits )
      output << 0;
   else
      for ( ; i < HugeInt::digits; i++ )
         output << num.integer[ i ];

   return output;
} // end function operator<<
```