Operator Overloading

Lab 8: Huge Integer

Rung-Bin Lin

International Program in Informatics for Bachelor
Yuan Ze University

04/07/2021

Objectives

- Discuss more advanced topics about operator overloading
  - Overloading as global functions
  - Overloading ++ and <=

Review of Operator Overloading (1)

- Operators that are overloaded as non-static member functions
  - The **leftmost operand** must be an object of the operator’s class.
    - An overloaded binary operator + used in **X+Y** will be transformed into a call **X.operator+(Y)**. Hence, **X** must be an object of the operator’s class. However, Y may be or may not be an object of the operator’s class.
- Operator precedence can not be changed by overloading
- No new operators can be created.

Restrictions on Operator Overloading (2)

- Operators that are overloaded as global functions
  - The leftmost operand may be an object of a different type or a fundamental type.
  - Like stream insertion operator << (or stream extraction operator >>), **cout << X,** if transformed into a function call, becomes **cout.operator<<(X)**. Since **cout** is not an object of the operator<<‘s class, we cannot implement operator<< as a member function of the object X’s class. Hence, operator<< should be implemented as a global function. Hence, **cout<<X** will be transformed into a call **operator<<(cout, X).**
  - Usually make friend to the class whose objects will use the operator.

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

- Modify the code for class HugeInt in Fig. 11.22 ~ Fig. 11.24 to make main() function work correctly. Here, it is assumed that all numbers are non-negative numbers.
  - Modify the class to process numbers with an arbitrarily finite number of digits. That is, there will be no overflow caused by any arithmetic operations. For this the private data of HugeInt class should be replaced by the following two variables:

    short *integer; // A dynamically allocated array for storing the digits of a HugeInt.

    int digits; // specified the number of digits in a HugeInt

    The size of an allocated array should be exactly the same as the number of digits in a HugeInt.
  - Overload operator+ to perform int + HugeInt and string + HugeInt. This may require overloading the operator as a global function.
  - Overload operator += to do HugeInt = HugeInt + int and HugeInt = HugeInt + anotherHugeInt.

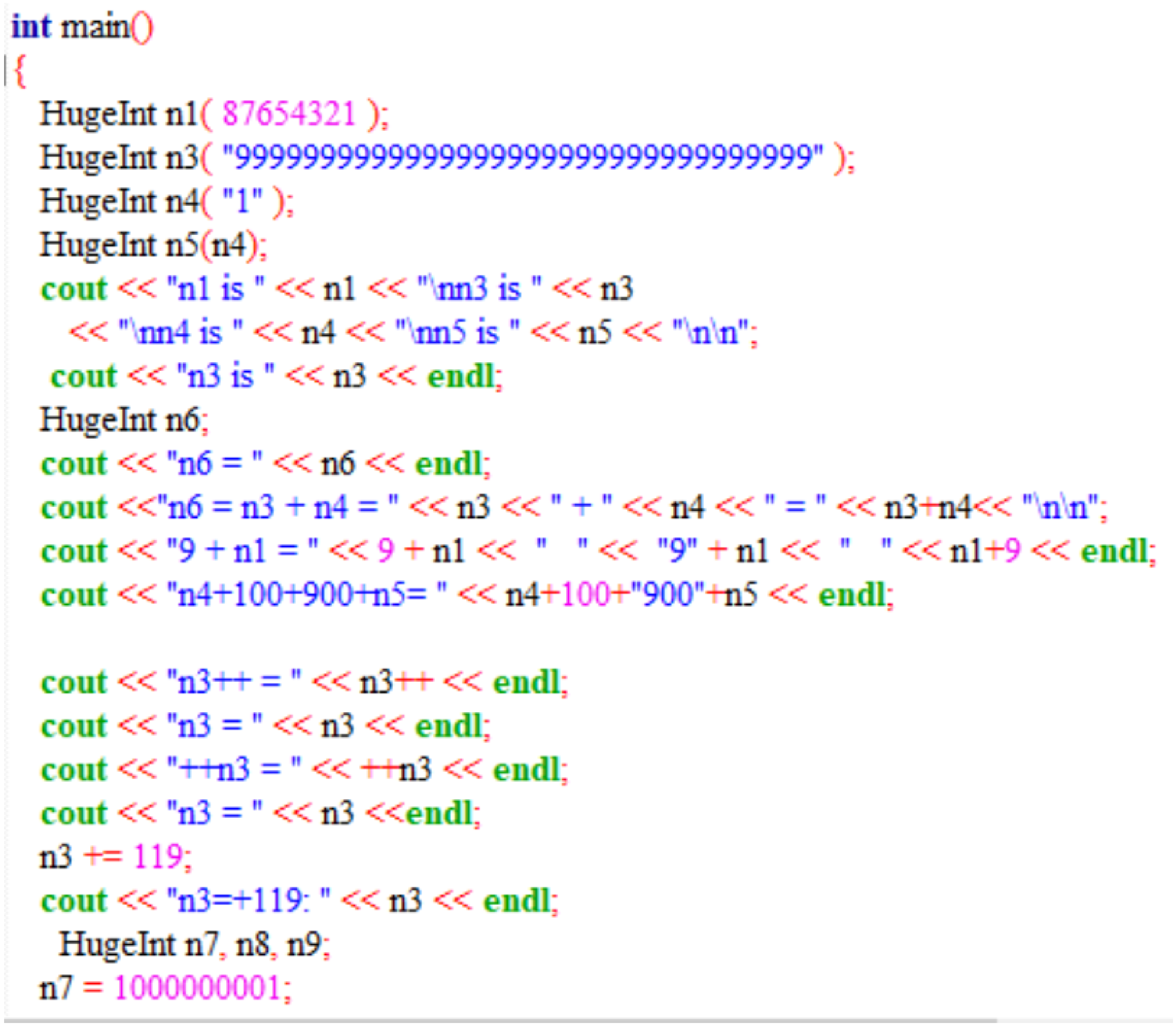
Lab 8 cont.

- 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.**
- **Overload the assignment operator= for doing HugeInt = HugeInt, HugeInt = int, and HugeInt = string.**
- **Implement a ~HugeInt() to delete the array allocated to a HugeInt. You should use delete[ ] to delete the memory held by a dynamically allocated array.**
- **Implement member function getNumDigits() that returns the number of digits in a HugeInt.**
- **The main() function is given and should not be modified.**
- **When printing out a HugeInt, you should also print out the number of digits enclosed within a pair of parentheses after the least significant digit. See output example for details.**

Discussions

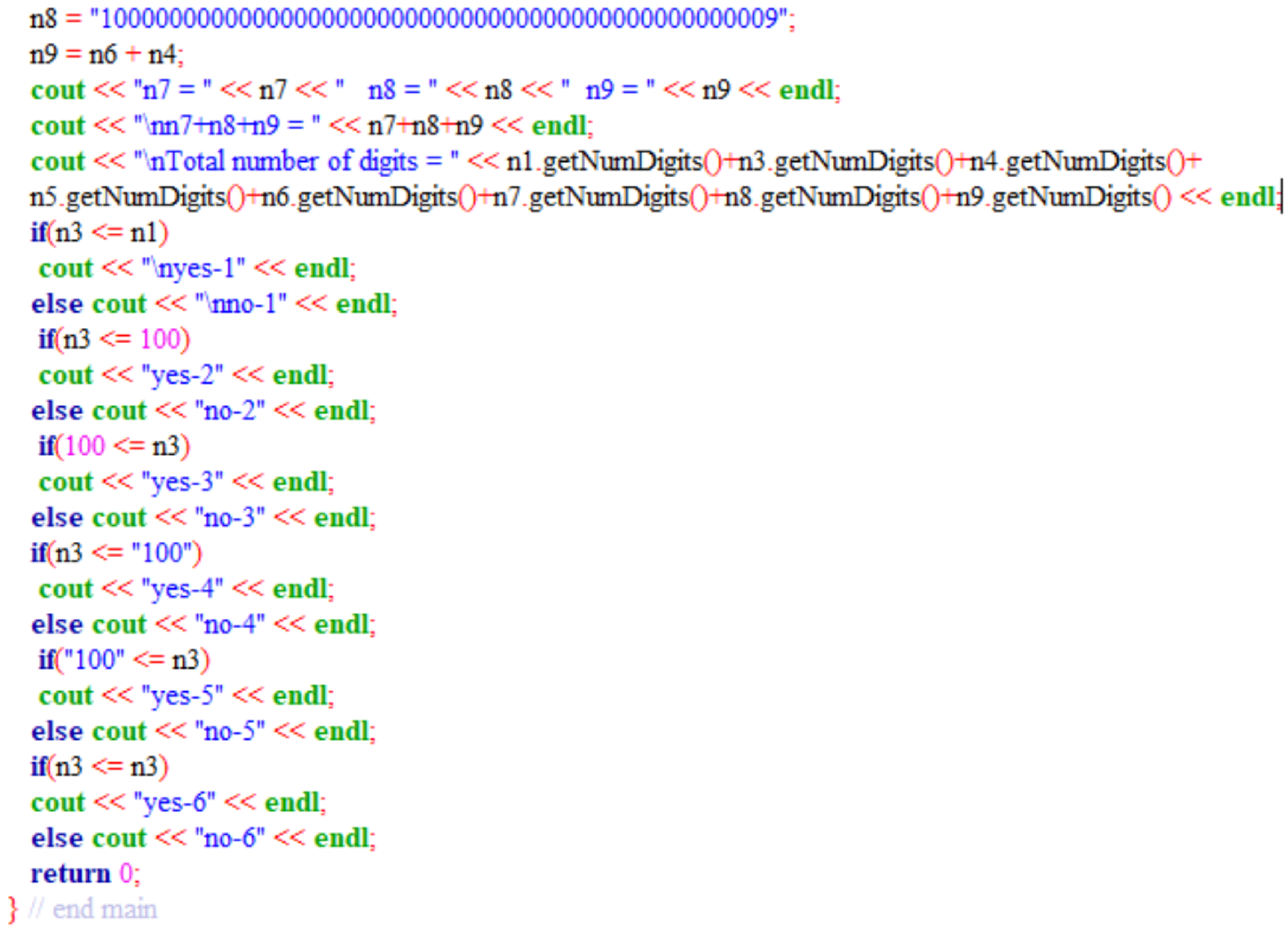
- Originally, the lab is designed to extend the HugeInt class to handle negative numbers and overload the operator-. However, it is found out that its implementation is quite complicate. Hence, the lab is modified to do what is presented here.
- To work out this lab, you should know how to use new and delete[ ] to perform dynamic memory allocation and deallocation. You need this for adjusting the array memory for storing the result of adding two numbers.
- When you implement a member function or a global function, you should always pay attention to the where the least significant digits of a HugeInt are located. Especially, when two HugeInts of different lengths are added.

Main() Function (1)



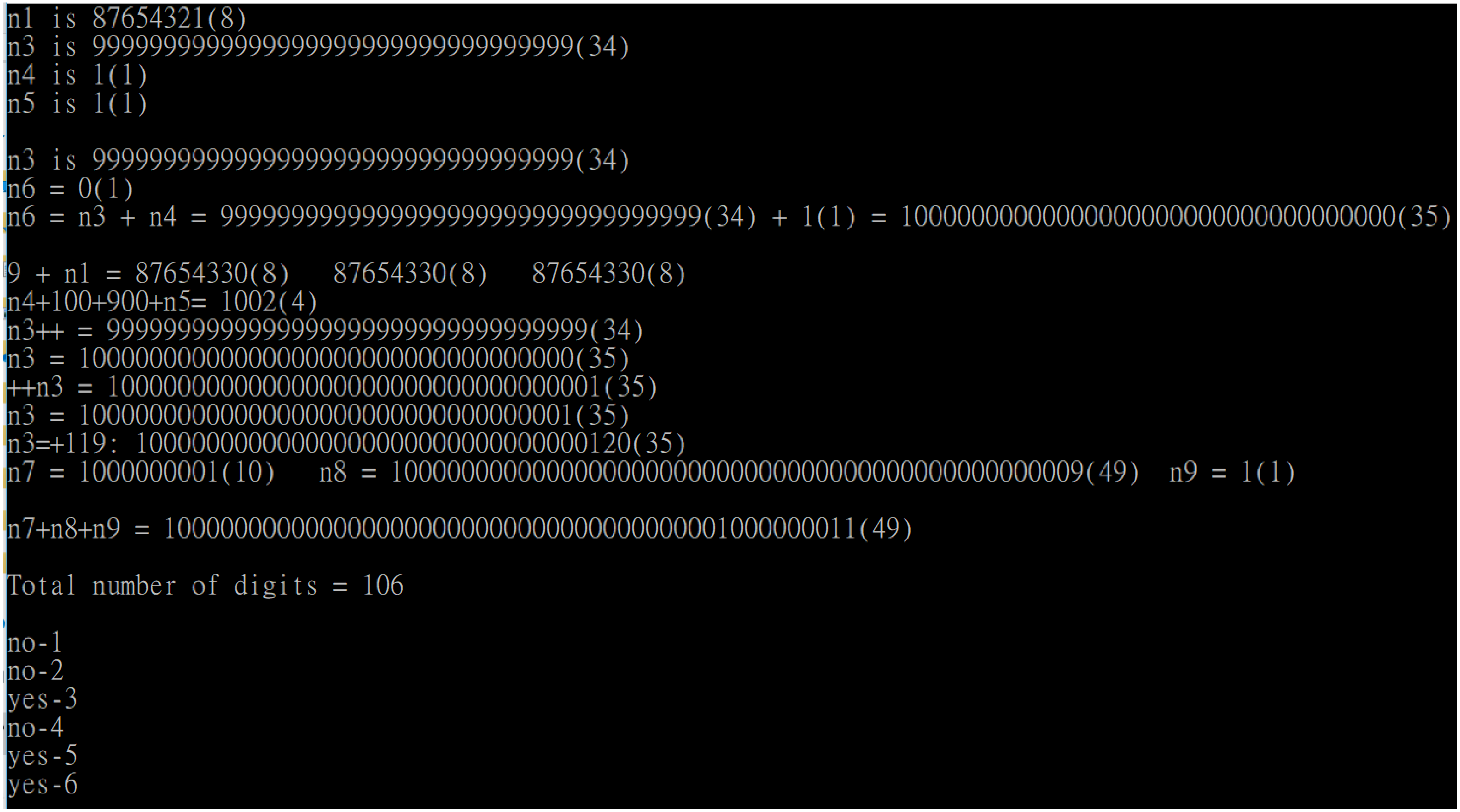
```
int main()
{
   HugeInt n1( 87654321 );
   HugeInt n3( "99999999999999999999999999999999999" );
   HugeInt n4( "1" );
   HugeInt n5(n4);
   cout << "n1 is " << n1 << "\nn3 is " << n3
      << "\nn4 is " << n4 << "\nn5 is " << n5 << "\n\n";
    cout << "n3 is " << n3 << endl;
   HugeInt n6;
   cout << "n6 = " << n6 << endl;
   cout <<"n6 = n3 + n4 = " << n3 << " + " << n4 << " = " << n3+n4<< "\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;
   n3 += 119;
   cout << "n3=+119: " << n3 << endl;
     HugeInt n7, n8, n9;
   n7 = 1000000001;
```

main() Function (2)



```
  n8 = "10000000000000000000000000000000000000000000000009";
  n9 = n6 + n4;
  cout << "n7 = " << n7 << "   n8 = " << n8 << "  n9 = " << n9 << endl;
  cout << "\nn7+n8+n9 = " << n7+n8+n9 << endl;
  cout << "\nTotal number of digits = " << n1.getNumDigits()+n3.getNumDigits()+n4.getNumDigits()+
  n5.getNumDigits()+n6.getNumDigits()+n7.getNumDigits()+n8.getNumDigits()+n9.getNumDigits() << 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;
  if(n3 <= n3)
  cout << "yes-6" << endl;
  else cout << "no-6" << endl;
  return 0;
} // end main
```

Output



```
n1 is 87654321(8)
n3 is 9999999999999999999999999999999999(34)
n4 is 1(1)
n5 is 1(1)

n3 is 9999999999999999999999999999999999(34)
n6 = 0(1)
n6 = n3 + n4 = 9999999999999999999999999999999999(34) + 1(1) = 10000000000000000000000000000000000(35)

9 + n1 = 87654330(8)   87654330(8)   87654330(8)
n4+100+900+n5= 1002(4)
n3++ = 9999999999999999999999999999999999(34)
n3 = 10000000000000000000000000000000000(35)
++n3 = 10000000000000000000000000000000001(35)
n3 = 10000000000000000000000000000000001(35)
n3=+119: 10000000000000000000000000000000120(35)
n7 = 1000000001(10)   n8 = 1000000000000000000000000000000000000000000000009(49)  n9 = 1(1)

n7+n8+n9 = 1000000000000000000000000000000000000001000000011(49)

Total number of digits = 106

no-1
no-2
yes-3
no-4
yes-5
yes-6
```

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.

```
// 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
```

```cpp
// 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
```

```cpp
// 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<<
```