CSCI 3110 Operator Overloading in C++

Operator overloading: give a different meaning to an existing operator by writing a function which redefines the operator. All operators can be overloaded except . .* :: ?: sizeof, A typical class should overload at least the following operators: =, ==, !=, <, <=, > >=

Rules of overloading an operator:

1. Overloaded operators can be used with a struct or a class.
2. The name of the function is operatorxx where xx is the operator to be overloaded
3. All operatorxx functions should have a return type.
4. You can overload operators as :

Client-defined external function

Example

```
<header file> struct
CardStruct
{
      SuitType     suit;       //the card's suit (hearts, spades, etc.)
      int value;               //the card's value (1 – 10, 11 = Jack, 12 = Queen, etc.)
 };

<implementation file>
bool operator == (const CardStruct & firstCard, const CardStruct & secondCard)
{
          return ((firstCard.suit == secondCard.suit) && (firstCard.value ==
                   secondCard.value));
}

<client file>
CardStruct      myCard, yourCard;
...
if (myCard == yourCard)
     cout << "What a coincidence!" << endl;
```

Struct/Class member

Example

```
<header file> struct
CardStruct
{
          SuitType     suit;
          int                    value;
          bool operator == (const CardStruct & rhs);
 };

<implementation file>
bool CardStruct :: operator == (const CardStruct & rhs)
{
          return ((suit == rhs.suit)&&(value==rhs.value));
}

<client file>
CardStruct      myCard, yourCard;
...
if (myCard == yourCard)
          cout << "What a coincidence!" << endl;
```

Friend

- Friend functions are not members of the class
- A friend function has access to private members of the class Friend functions are declared with the reserve word friend.
- Use friend functions sparingly
- Examples where they might be used are:

Overload the insertion operator << and extraction operator >>

Example:
To overload << and >> in a class called PlayerClass

```
class PlayerClass
{
public:
. . .
    PlayerClass& operator = (const PlayerClass& rhs);

    // version 1: define the code in the class definition header file directly
    friend istream& operator >> (istream& is, PlayerClass& rhs) {
        is >> rhs.name >> rhs.currentScore;
        return is;
    }

    friend ostream& operator << (ostream& os, const PlayerClass& rhs) {

        os << rhs.name << "'s current score is "
            << rhs.currentScore << endl;
        return os;
    }

private:
    string name;
    int     currentScore;
};

PlayerClass & PlayerClass::operator = (const PlayerClass & rhs)
{
    name = rhs.name;
    currentScore = rhs.currentScore;
    return *this;
}
```

<In the client file>
The overloaded functions will be called using statements similar to the following:

```
PlayerClass playerA, playerB;
ifstream myin;
ofstream myout;
…
myin>> playerA;         // or cin>>playerA;
myout<<playerB;         // or cout << playerB;


playerB = playerA;    // assignment operator
```

```
// version 2: define the >> and << operators in the implementation file
    class PlayerClass
    {
    public:
    . . .
        PlayerClass& operator = (const PlayerClass& rhs);

        // version 2: declare the friend function prototype in the class header file
        friend istream& operator >> (istream& is, PlayerClass& rhs);
        friend ostream& operator << (ostream& os, const PlayerClass& rhs);

    private:
        string name;
        int     currentScore;
    };
```

In the implementation file, i.e., playerClass.cpp:

```
// notice there is no class name quantification and no keyword “friend” here:
istream& operator >> (istream& is, PlayerClass& rhs) {
    is >> rhs.name >> rhs.currentScore;
    return is;
}
ostream& operator << (ostream& os, const PlayerClass& rhs) {

    os << rhs.name << “’s current score is “
        << rhs.currentScore << endl;
    return os;
}
```

Example of overloaded operators used with ListClass (Array implementation)

Overloading operators (==,=,   [ ] subscript, + meaning concatenation of two lists and friend functions (<< and >>) for the array based list class.

```
//*****************************************
// Header file List.h for the ADT list (Array-based implementation)
//*****************************************
const int MAX_LIST =200;
typedef int ListItemType;    // can be easily changed to ADT of other types

class List
{
public:
  List(); // default constructor
        // destructor is supplied by compiler

  // list operations:
  bool isEmpty() const;
  // Determines whether a list is empty.
  // Precondition: None.
  // Postcondition: Returns true if the list is empty;
  // otherwise returns false.

  int getLength() const;
  // Determines the length of a list.
  // Precondition: None.
  // Postcondition: Returns the number of items
  // that are currently in the list.

  void insert(int index, ListItemType newItem, bool& success);
  // Inserts an item into the list at position index.
  // Precondition: index indicates the position at which the item should be inserted in the list.
  // Postcondition: If insertion is successful, newItem is  at position index in the list, and other items are
  // renumbered accordingly, and success is true; otherwise success is false.
  // Note: Insertion will not be successful if  index < 1 or index > getLength()+1.

  void remove(int index, bool& success);
  // Deletes an item from the list at a given position.
  // Precondition: index indicates where the deletion should occur.
  // Postcondition: If 1 <= index <= getLength(),
  // the item at position index in the list is deleted, other items are renumbered accordingly,
  // and success is true; otherwise success is false.

  void retrieve(int index, ListItemType& dataItem, bool& success) const;
  // Retrieves a list item by position.
  // Precondition: index is the number of the item to be retrieved.
  // Postcondition: If 1 <= index <= getLength(), dataItem is the value of the desired item and
  // success is true; otherwise success is false.

private:
    ListItemType items[MAX_LIST];      // array of list items
```

```
      int         size;          // number of items in list

    int translate(int index) const;
    // Converts the position of an item in a list to the correct index within its array representation.
}; // end List class

//******************************************
// Implementation file List.cpp for the ADT list (Array-based implementation)
//******************************************
#include "List.h" //header file

List::List() : size(0) {
} // end default constructor

bool List::isEmpty() const {
        return bool(size == 0);
} // end isEmpty

int List::getLength() const {
        return size;
} // end getLength

void List::insert(int index, ListItemType newItem, bool& success) {
        success = bool( (index >= 1) && (index <= size+1) && (size < MAX_LIST) );
        if (success)    {  // make room for new item by shifting all items at
                // positions >= index toward the end of the list (no shift if index == size+1)
                for (int pos = size; pos >= index; --pos)
                        items[translate(pos+1)] = items[translate(pos)];

                // insert new item
                items[translate(index)] = newItem;
                ++size; // increase the size of the list by one
        } // end if
} // end insert

void List::remove(int index, bool& success) {
        success = bool( (index >= 1) && (index <= size) );

        if (success)    {
        // delete item by shifting all items at positions > index toward the beginning of the list
        // (no shift if index == size)
                for (int fromPosition = index+1;  fromPosition <= size; ++fromPosition)
                        items[translate(fromPosition-1)] =  items[translate(fromPosition)];
                --size; // decrease the size of the list by one
        } // end if
} // end remove

void List::retrieve(int index, ListItemType& dataItem,  bool& success) const
{
        success = bool( (index >= 1) &&  (index <= size) );
        if (success)
                dataItem = items[translate(index)];
```

```
} // end retrieve

int List::translate(int index) const {
        return index-1;
} // end translate
```

Below are overloaded operators added to the listClass :

1. **Add the declarations of the overloaded operators to the header file:**

```
class List
{
public:
        … // other methods already declared

          // overloaded operators declared
        bool operator == (const listClass & rhs);
        listClass & operator = (const listClass & rhs);
        listClass  operator + (const listClass & rhs);
        listItemType& operator[]( unsigned nSubscript );

        friend ostream & operator << (ostream & os, const listClass & rhs);
};
```

2. **Define these operators in the implementation file:**

```
bool listClass::operator == (const listClass & rhs)
{
    if (size != rhs.size)
          return false;
    else    {
          for (i=0; i<size; i++) {
                if (items[i] != rhs.items[i])
                      return false;
          }
          return true;
    }
}

listClass & listClass::operator = (const listClass & rhs)
{
      listItemType newItem;

      if (this != &rhs) {
          size = rhs.size;

            for (int i=0; i<rhs.size; i++) {
                items[i] = rhs.items[i];
            }
    }

    return (*this);
}
```

```
// do not use listClass & return type because :
// (1) it makes (l1+l2) = l3 legal,
// (2) “list”, as a local variable, is de-allocated after the function exits.
// Referencing to such a variable is not safe.

listClass  listClass :: operator + (const listClass & rhs) {
        int             leftSize = size;
        listClass    list(*this);    // assuming we have a copy constructor available

        // assuming there is enough memory in the array to hold the concatenation of the two
lists
          for (int i=1; i<=rhs.size; i++)         {
               list.ListInsert(leftSize+i, rhs.items[i-1], success);
               size ++;
        }
        return list;
}


 // friend function : overloaded << operator
ostream & operator << (ostream & os, const listClass & rhs) {
          os << “ The list of items are: “ << endl;
          for (int i=0; i<rhs.size; i++)
                    os << rhs.items[i] << endl;
          return os;
}


listItemType& listClas::operator[]( unsigned nSubscript ) {

          listItemType  item;
          bool  success;

          if( nSubscript < size){
                     listRetrieve(nSubscript+1, item, success);
                    if (success)
                              return item;
                    else {
                              cerr << “error retrieving the “ << nSubscript << “th item” << endl;
                              exit(-1);     // exception handle in the future
                    }
          }
          else {
                    cerr << "Array bounds violation." << endl;
                    exit(-1);   // exception handle in the future
          }
}
```

```
//******************************************
//   Client Program using ADT list
//******************************************
#include “List.h”
#include <iostream>
using namespace std;

int main() {
      List                  aList;
      ListItemType    item;
      bool                  success;

      for (int i=1; i<=MAX_LIST; i++)       {
            cout << “Enter list item “ << i << endl;
            cin >> item;
            aList.insert(i,  item, success);
      }

       PrintInReverse(aList);
       SortList(aList);

       return 0;
}
```