COMP 151H: Object Oriented Programming (Honor Track)
Spring Semester 2006
Midterm Exam
Instructor: Chi Keung Tang
Wednesday, March 15, 2006
7:00 – 9:00pm
Room 2502

This is a **CLOSED-BOOK-CLOSED-NOTES** exam consisting of three (3) problems. Follow the instructions carefully. Please write legibly in the space provided. Keep the exam booklet stapled.

K E Y

Problem	Points	your score
1 INFORMATION HIDING	4	
2 CONTAINER CLASS	8	
3 OBJECT CONSTRUCTION AND DESTRUCTION	3	
Total	15	

1 Information hiding

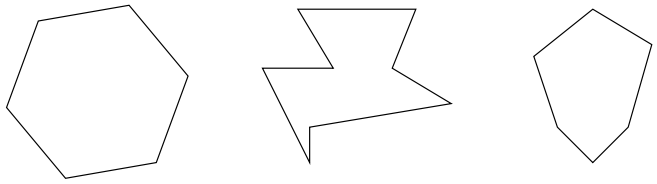

Figure 1: Polygon examples

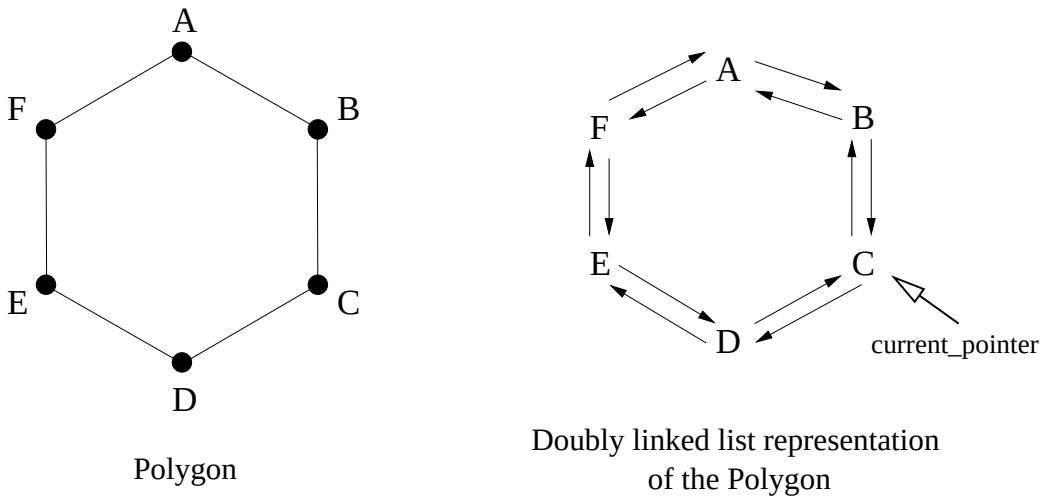

Figure 2: Polygon representation

In this question, you are required to implement two member functions for a class **Polygon** to manipulate a 2D polygon. For example, Figure 1 shows some typical polygons. Actually, a polygon can be represented by a circular doubly linked list (see Figure 2). An edge connecting a node (Point) A and B is represented by a next-pointer in node A and a previous-pointer in node B.
The **Polygon** class contains a circular doubly linked list that stores the points or vertices of a polygon.
Here is the structure of a `Point` for a 2D point or vertex:

```
struct Point{
     int x;            // x-coordinate
     int y;            // y-coordinate
}
```

And, here is the definition of the **Polygon** class:

```
class Polygon {
public:
  Polygon();    // default constructor
  ~Polygon();   // destructor
  bool isCollide ( Polygon& inPolygon);  // returns true if the inPolygon
                                         // collides with this Polygon;
                                         // returns false if otherwise

// setPolygon takes an array of points and forms a polygon
  void setPolygon (Point pts[], int size)
  {
    vertexList.clear();
    for (int i=0; i<size; i++)
    {
       vertexList.insertToNext ( pts[i] );
       vertexList.pointToNext ();
    }
  }
// Two member functions to be implemented by you
  Polygon* cutPolygon();
  bool trimPolygon ( Polygon& inPolygon );

private:

  LinkedList vertexList;      // The circular doubly linked list
};
```

As you can see, the polygon class contains a private variable called **vertexList** which is a **LinkedList** object. This is the circular doubly linked list. The definition of the **LinkedList** is defined as follow:

```
class LinkedList{
public:

    LinkedList();          // default constructor
    ~LinkedList();         // destructor

    int  getSize() const;       // return the number of elements(node) of the linked list
    bool isEmpty() const;       // return true if the list is empty
    void clear();               // make the circular doubly linked list empty
    void deleteCurrentNode();   // delete the current node. The current pointer will
                                // point to the next node of the deleted node
    void pointToNext();         // make the current_pointer point to the next node
    void pointToPrev();         // make the current_pointer point to the previous node
    Point getCurrentPoint() const;          // return the Point pointed by the current_pointer
    void  insertToNext( const Point& ptr );  // insert a Point next to the current node
    void  insertToPrev( const Point& ptr );  // insert a Point before the current node

private:

    // The current pointer. It points to the current node.
    // If the linked list is empty, it is equals to NULL.
    Node* current_pointer;

    // Other member variables are skipped
    // ...

};
```

The functionalities of the member functions of **LinkedList** are stated in the comments in the class definitions. In this question, it is assumed that the circular doubly linked list has been implemented for you. So, you are free to use its member functions (without the need to know their implementation details).

Be careful! Except for `cutPolygon()` and `trimPolygon()` to be implemented, you must not use member (and non-member functions) other than those declared in this question.

(a) *(2 points)* Implement the member function

$$Polygon* \ Polygon :: cutPolygon()$$

This function cuts a corner triangle from the original (**this**) polygon. Let N be the size of the original polygon which is referred to by the **this** pointer.

- If $N \leq 3$, this function returns NULL immediately in order to prevent error.
- Otherwise, the function will copy three consecutive vertices from **this** polygon (starting from the current pointer position of the **vertexList**). These three consecutive vertices will form the corner triangle to be cut (by connecting the first and last node) and be returned as the return argument.
- Given the three consecutive vertices of the corner triangle to be cut, the first and the last node will be kept while the other will be deleted from **this** Polygon. The first and the last node will be connected.
- Finally, **this** polygon is modified as the one after cutting the corner triangle.

Implement the above pseudo-code. Write your code clearly here:

```
Polygon* Polygon::cutPolygon()
{
        int polygonSize = vertexList.getSize();
        if( polygonSize <= 3 ) return NULL;

        Point* consecutivePt = new Point[ 3 ];

        for( int i=0; i<3; i++)
        {
                consecutivePt[i] = vertexList.getCurrentPoint();
                if( i==0 || i==2 )
                        vertexList.pointToNext();
                else
                        vertexList.deleteCurrentNode();
        }

        Polygon* newPolygon = new Polygon;
        newPolygon->setPolygon( consecutivePt, 3 );

        delete [] consecutivePt;

        return newPolygon;
}
```

(b) *(2 points)* Implement the member function

$$bool\ Polygon :: trimPolygon(Polygon\&\ inPolygon)$$

This function repeatedly trims (and therefore modifies) the `inPolygon` by calling `cutPolygon` until it does not collide with **this** Polygon and returns true. Else, if the `inPolygon` cannot be trimmed further and yet it still collides with **this** Polygon, the function returns false.

Implement the following pseudo-code:

- If the size of the `inPolygon` is less than or equal to 3, return false because `inPolygon` is a degenerate polygon.
- The function iteratively checks, using `isCollide`, if the `inPolygon` collides with **this** Polygon, and returns true *as soon as* no collision occurs. Else, use `cutPolygon` to reduce the size of the `inPolygon`.
- The function returns false if the `inPolygon` cannot be further cut while it still collides with **this** Polygon.
- You need to eliminate all the memory leak in the implementation.

```
bool Polygon::trimPolygon ( Polygon& inPolygon )
{
        int polygonSize = inPolygon.vertexList.getSize();
        if ( polygonSize <= 3 ) return false;

        while (isCollide(inPolygon))
        {
                Polygon * tri = inPolgon.cutPolygon ();
                if (!tri) { delete tri; return false; }
                // inPolygon cannot be trimmed further
                delete tri;             // avoid memory leak
        }
        return true; // trimmed inPolygon  does not collide with this Polygon
}
```

2 Container class

This question concerns with the design and implementation of a container class `Array`, where you should apply what you learned about templates and operator overloading.
Note that your implementation should be free of any dangling pointers/references or memory leaks in order to avoid mark deduction.
(8 points) Code up `array.h` so that the following main program can compile. The sample input and output are shown.

```
//-----------------------main.cpp---------------------------------
#include <iostream>
#include "array.h"
using namespace std;

int main()
{
  int i, size;
  cout << "Enter size : "; cin >> size;
  Array<char> a(size);
  // ASCII code for 'A' is 65.
  for (i=0;i<6;i++) { a[i] = i+65; cout << (char)(a[i]) << " " ; }
  cout << endl;

  Array<char> b(a);
  for (i=0;i<size;i++) { cout << (char)(b[i]) << " "; }
  cout << endl;
  cout << ((a.getsize() == b.getsize()) ? "true" : "false") << endl;
  cout << a[100] << endl;  // out of bound
  b[10000] = 'j';          // out of bound
  for (i=0;i<size;i++) {  // the array b is unchanged
   cout << (char)(b[i]) << " ";
  }
  cout << endl;

  Array<char> c, d;
  d = c = b;  cout << "cascading assignment..." << endl;
  for (i=0;i<d.getsize();i++) d[i] += 1;
  for (i=0;i<d.getsize();i++) cout << d[i] << " "; cout << endl;
  for (i=0;i<c.getsize();i++) c[i] += 2 ;
  for (i=0;i<c.getsize();i++) cout << c[i] << " "; cout << endl;
  for (i=0;i<b.getsize();i++) cout << b[i] << " ";
  cout << "<--- b remains unchanged..." << endl;

  return 0;
}
```

Sample input and output:

```
Enter size : 4
A B C D array index 4 out of bound - value UN trustworthy
array index 4 out of bound - value UN trustworthy
E array index 5 out of bound - value UN trustworthy
array index 5 out of bound - value UN trustworthy
F
A B C D
true
array index 100 out of bound - value UN trustworthy
F
array index 10000 out of bound - value UN trustworthy
A B C D
cascading assignment...
B C D E
C D E F
A B C D <--- b remains unchanged...
```

```
#ifndef ARRAY_H
#define ARRAY_H

template <typename T>
class Array {
public:
   Array (int = 10);
   Array (const Array &);
   T& operator[](int i);
   const T& operator[](int i) const;
   Array& operator= (const Array &);
   int getsize() const { return _size; };
   ~Array();
private:
   T* _data;
   int _size;
};

template<typename T>
Array<T>::Array (int size) : _size(size) {
  _data = new T[size];
  if (!_data) {
    cerr << "Memory Error" << endl;
    exit(1);
  }
}

template<typename T>
Array<T>::Array (const Array<T> &a) : _size (a._size) {
  _data = new T[a._size];
  if (!_data) {
    cerr << "Memory Error" << endl;
    exit(1);
  }
  memcpy(_data,a._data,_size*sizeof(T));
}

template<typename T>
Array<T>::~Array () {
  if (_data) {
    delete [] _data;
    _data = NULL;
    _size = 0;
  }
}

template<typename T>
T& Array<T>::operator[](int i) {
  static T tmp = 'Z';
  if ((i < 0) || (i >= _size)) {
    cerr << "array index " << i << " out of bound - value UN trustworthy" << endl;
    return tmp; // return a dummy reference
  } else
    return _data[i];
}

template<typename T>
const T& Array<T>::operator[](int i) const {
  static T tmp = 'Z';
  if ((i < 0) || (i >= _size)) {
    cerr << "array index " << i << " out of bound - value UN trustworthy" << endl;
```

```
    return tmp; // return a dummy value
  } else
    return _data[i];
}

template<typename T>
Array<T>& Array<T>::operator= (const Array &a) {
  _size = a._size;
  if (_data) delete [] _data;    // free the old memory if it exists
  _data = new T[a._size];
  if (!_data) {
    cerr << "Memory Error" << endl;
    exit(1);
  }
  memcpy(_data,a._data,_size*sizeof(T));
  return *this;
}

#endif
```

3 Object construction and destruction

This problem requires you to apply what you learned on the order of construction and destruction. An object of another class, which can be static or otherwise, is one data member of the enclosing class.
Consider the following program:

```
//---------------------obj.cpp--------------------
#include <stdio.h>
class A {
public:
  A() { a = new int ; puts ("A created"); }
  ~A() { delete a; puts ("A destroyed"); }
private:
  int *a;
};
class B {
public:
  B() { aa  = new A ; puts ( "B created");}
  ~B() { delete aa ; puts ( "B destroyed");}
private:
  A sa;   // <---- (*)
  A* aa;
};
int main()
{
  B tryB;
  B *dynB = new B;
  delete dynB;
  return 0;
}
```

(a) *(1.5 points)* Give the output of the program.

```
A created
A created
B created
A created
A created
B created
A destroyed
B destroyed
A destroyed
A destroyed
B destroyed
A destroyed
```

(b) *(1.5 points)* The statement marked by `(*)` is replaced by `static A sa`. The program is then recompiled and run. Give the output of the new program.

```
A created
B created
A created
B created
A destroyed
B destroyed
A destroyed
B destroyed
```