CS 151: Object Oriented Programming
Spring Semester 2004
Midterm Exam
Instructor: Chi-Keung Tang
Thursday, March 4, 2004
7:00 – 9:00pm
at LTB or LTC

This is a **CLOSED-BOOK-CLOSED-NOTES** exam consisting of two (2) problems. Follow the instructions carefully. Please write legibly in the **boxes** provided. *Any space outside the boxes are for sketching and will not be graded.* Keep the exam booklet stapled.
Hint: THINK CAREFULLY! All correct answers are SIMPLE, but not necessarily easy.

# K E Y

| Problem | Points | your score |
|---|---|---|
| **1** POLYGON | 5 | |
| **2** REGISTER | 10 | |
| Total | 15 | |

# 1 Polygon

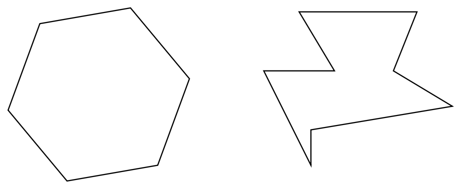

Figure 1: Polygon examples

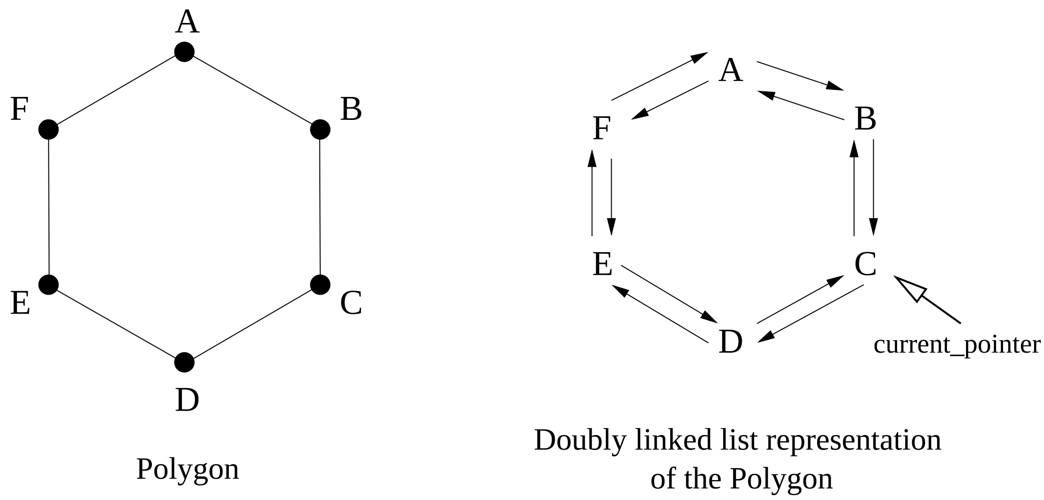


Figure 2: Polygon representation

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.
As the result, 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

    // It take an array of points and form a polygon
    void setPolygon( Point pts[], int size)
    {
        vertexList.clear();
        for( int i=0; i<size; i++ )
        {
            vertexList.insertToNext( pts[i] );
            vertexList.pointToNext();
        }
    }

    // To be implemented
    Polygon* splitPolygon();

    // To be implemented
    bool isCollide( Polygon& inPolygon );

    // The input edge is defined by 2 end points - ptA and ptB
    // This function return true if the input edge intersect
    // or touch this polygon. Otherwise, it return false.
    bool isEdgeIntersect( const Point& ptA, const Point& ptB );

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 `splitPolygon()` and `isCollide()` to be implemented, you can use, and can *only* use, the member functions declared in this question.

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

$$bool\ Polygon :: isCollide(Polygon\&\ inPolygon);$$

This function checks whether **this** polygon collides with **inPolygon** or not. If collision occurs, this function return **true**. Otherwise, it return **false**.

To know whether **this** polygon collides with **inPolygon**, one simple way is:

- If one of the polygon contains no vertices, return **false** because there must be no collision.
- For each edge in **inPolygon**, test whether the edge intersects or touches **this** polygon.
- If one or more edges of **inPolygon** intersect or touch **this** polygon, collision occurs and you can return **true** immediately.
- Otherwise, collision does not occur.

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

```
bool Polygon::isCollide( Polygon& inPolygon )
{
        if( vertexList.isEmpty() ) return false;
        if( inPolygon.vertexList.isEmpty() ) return false;

        bool notCollide = true;
        int inPolySize = inPolygon.vertexList.getSize();
        for( i=0; i<inPolySize; i++)
        {
                Point ptA, ptB;
                ptA = inPolygon.vertexList.getCurrentPoint();
                inPolygon.vertexList.pointToNext();
                ptB = inPolygon.vertexList.getCurrentPoint();

                if( this->isEdgeIntersect( ptA, ptB ) )
                {
                        notCollide = false;
                        break;
                }
        }

        return (!notCollide);
}
```

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

$$Polygon* \; Polygon :: splitPolygon()$$

This function splits the original (**this**) polygon into 2 polygons by the followings:

- Let the size of the original polygon be $N$. The function will copy $M = floor(N/2) + 1$ consecutive vertices from the **this** polygon (starting from the current pointer position of the **vertexList**). These $M$ consecutive vertices will form a new polygon (by connecting the first and last node) and be returned as the return argument.
- Among the $M$ consecutive vertices in the **this** polygon, except the first and the last node, these consecutive vertices will be deleted. And, the first and the last node will be connected.
- Finally, **this** polygon is modified. You should get a new polygon from the return argument.
- When $N \leq 3$, this function returns NULL immediately, in order to prevent error.

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

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

        int mSize = polygonSize/2 + 1;
        Point* consecutivePt = new Point[ mSize ];

        for( int i=0; i<mSize; i++)
        {
                consecutivePt[i] = vertexList.getCurrentPoint();
                if( i==0 || i==mSize-1 )
                        vertexList.pointToNext();
                else
                        vertexList.deleteCurrentNode();
        }

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

        delete [] consecutivePt;

        return newPolygon;
}
```

(c) *($\frac{1}{2}$ point)* If you come to this part successfully, you may find an interesting thing that you can implement the code of **Polygon** without the knowledge of the implementation and private data members of the **LinkedList** class. Now, if I tell you that I want to use an array to implement the **LinkedList** class, you just need to modify the private data members and the member function implementation of **LinkedList** such that we change nothing in the code of the **Polygon** class. In the view of implementation, it is achieved by defining a **Class** and hiding data members and function implementation.

State the keywords in **no more than 2 words** the object oriented programming concept you learned in comp151 related to the above.

`information hiding / data encapsulation`

(d) *($\frac{1}{2}$ point)* As you can see, the **LinkedList** class is specified for **Point** structure. It is not a good practice because you have to "copy-and-paste" your code many times if you have to use many linked lists with different data types.

In C++, generic programming allows us to generalize a class for different data type.

State the most important C++ preserved word as the key word of this feature. **Only one word is allowed**.

`template`

## 2 Hash Table

Assume the template class *DoublyLinkedList* is already defined and implemented. Its class definition is shown below:

```
template<class ElemType>       // ElemType must define the operator<
class DoublyLinkedList
{
    public:
        DoublyLinkedList();
        ~DoublyLinkedList();
        DoublyLinkedList(const DoublyLinkedList& copy);
        DoublyLinkedList& operator=(const DoublyLinkedList& assign);

        // insert an element
        void insert(const ElemType& element);

        // clear the whole list
        void clear();

        // other member functions ...

    private:
        // other members ...
};
```

Also, the template class *HashTable* is also defined in HashTable.h

```
#include "List.h"

template<class ElemType>        // ElemType must define the operator<
class HashTable
{
    public:
        typedef unsigned int (*HashFunc) (const ElemType&);

        // hashFunc must return an integer in-between 0 and (numBuckets-1)
        HashTable(int numBuckets, HashFunc hashFunc);    // (1)

        HashTable(const HashTable& copy);                // (2)
        HashTable& operator=(const HashTable& assign);
        ~HashTable();                                    // (3)

        // insert an element
        void insert(const ElemType& element);

        // other member functions

    private:
        DoublyLinkedList<ElemType>* m_buckets;  // buckets
        unsigned int    m_numBuckets;           // number of buckets
        HashFunc        m_hashFunction;         // hash function
};
```

Notice: You **must** answer this question by making use of the above definition. Do not define your own linked list and hash table. However, you are free to use any member functions defined in class *DoublyLinkedList*.

(a) *($\frac{1}{2}$ point)* What is the type represented by *HashFunc* in the template class *HashTable* after template instantiation:

(1) with template argument ElemType = int

(2) with template argument ElemType = student, a user-defined class

```
unsigned int (*) (const int&)

unsigned int (*) (const student&)
```

(b) *(1 point)* **Circle** the statement below which results in compilation error. Explain briefly the cause of the compilation error.

```
#include "HashTable.h"
int main()
{
    HashTable<int> iHashTable;    // <----- this statement
    iHashTable.insert(10);
    iHashTable.insert(20);
    return 0;
}
```

Cause of compilation error: `Default constructor is not provided and generated in the template class HashTable.`

(c) *(5.5 points)* Implement the typical constructor, copy constructor and destructor for the template class *HashTable*. That is, implement member functions (1), (2), and (3). Write your answer on the next page.

```
template<class ElemType>
HashTable<ElemType>::HashTable<ElemType>(int numBuckets, HashFunc hashFunc)
{
m_buckets = new DoublyLinkedList<ElemType>[numBuckets];
if ( m_buckets == NULL ) exit(-1);

m_numBuckets = numBuckets;
m_hashFunction = hashFunc;
}

template<class ElemType>
HashTable<ElemType>::HashTable<ElemType>(const HashTable<ElemType>& copy)
{
m_numBuckets = copy.m_numBuckets;
m_buckets = new DoublyLinkedList<ElemType>[m_numBuckets];
if ( m_buckets == NULL ) exit(-1);

for (unsigned int i=0; i<m_numBuckets; i++)
m_buckets[i] = copy.m_buckets[i];
m_hashFunction = copy.m_hashFunction;
}

template<class ElemType>
HashTable<ElemType>::~HashTable<ElemType>()
{
if (m_buckets != NULL)
delete [] (m_buckets);
}
```

(d) *(2 points)* Implement the member function `insert` for the template class *HashTable*.

```
template<typename ElemType>
void HashTable<ElemType>::insert(const ElemType& element)
{
unsigned int hashValue = m_hashFunction(element);
m_buckets[hashValue].insert(element);
}
```

For the part (e) and part (f) below, we only consider the class *DoublyLinkedList*. Let the class *DoublyLinkedList* be defined in a header file (List.h) and implemented in a source file (List.cpp). Consider the following main program (main.cpp):

```
#include "List.h"
int main()
{
    DoublyLinkedList<int> iList;
    iList.insert(3);
    return 0;
}
```

(e) *($\frac{1}{2}$ point)* After *g++ -c List.cpp* is executed, the compiled object file List.o is **very small** (even smaller than an object file for a simple "Hello World" program). Note that there is no `#include "List.cpp"` in `List.h`. Can you think of the possible reason? (Hint: the answer is very simple, pertaining to the idea of templates in C++.)

```
No object code will be generated for a non-instantiated template class.
```

(f) *($\frac{1}{2}$ point)* Consider the following Makefile:

```
all: main.o List.o
    g++ -o main main.o List.o
main.o: main.cpp List.h
    g++ -c main.cpp
List.o: List.h List.cpp
    g++ -c List.cpp
```

After *gmake* is executed, there are many linking errors complaining about undefined or unresolved symbols. Explain the reason briefly.

Hint:

(1) This question is closely related to the answer of part (e).

(2) Your answer should state clearly what is defined in each of the two object files (List.o and main.o) in order to get full credit.

```
From part (e), the object file (List.o) will not consist of any
instantiated DoublyLinkedList.

In the compiled main.o, the compiler has instantiated the
template class DoublyLinkedList's definition only. Its member
function (say "insert") is defined in the List.cpp, but the
compiler will not read the List.cpp when main.o is being
compiled. Therefore, the object file (main.o) has no
definition of its member functions.

In the linking step, it tries to link the two object files
together. But the linker cannot find the definition of the
member functions of the class DoublyLinkedList (with template
argument ElemType = int). Therefore, it results in
"unresolved/undefined symbols".
```