UCLA Computer Science Department Instructor: Kung-Hua Chang

Student Name and ID _____

CS 32, SUMMER 2015, PRACTICE MIDTERM I.

Problem #	Maximal Possible Points	Received
1.1	1	
1.2	3	
1.3	3	
1.4	3	
1.5	3	
1.6	3	
2	4	
3.1	2	
3.2	2	
3.3	2	
3.4	3	
3.5	3	
3.6	3	
Total	35	

Problem #1: Below is a definition of a LinkedList class and a test main program:

```
#include <iostream>
using namespace std;
class LinkedList
{
public:
	LinkedList(): head(nullptr) { }
	~LinkedList();
	void addToList(int value);    // add to the end of the linked list
	void reverse();                    // Reverse the linked list
	void output();
	int JosephusCircle(int N);
private:
	struct Node
	{
		int num;
		Node *next;
	};
	Node *head;
};
void LinkedList::output()
{
	Node *ptr = head;
	cout << "The elements in the list are: ";
	while(ptr!=nullptr) {
		cout << ptr->num << " ";
		ptr = ptr->next;
	}
	cout << endl;
}
int main()
{
	LinkedList list;
	for(int i=1;i<=10;i++)	list.addToList(i);
	list.output();
	list.reverse();
	list.output();
}
```

Problem #1.1: Please complete the implementation of the **destructor**.

```
LinkedList::~LinkedList()
{
	Node *temp;
	while(head != nullptr)
	{
		temp = head;
		head = head->next;
		delete temp;
	}
}
```

Problem #1.2: Please write an implementation of the **addToList** member function.

```
void LinkedList::addToList(int value)
{
	Node *current = new Node;
	current->num = value;
	current->next = nullptr;
	if( head == nullptr)
		head = current;
	else
	{
		Node *ptr = head;
		while(ptr->next != nullptr)
			ptr = ptr->next;
		ptr->next = current;
	}
}
```

Problem #1.3: Please complete the implementation of the **reverse** member function.

```
void LinkedList::reverse()
{
    Node *nextNode = nullptr,*prevNode = nullptr, *current = head;
    while(current != nullptr ) {
    // Hint: Only 4 lines of codes are needed inside the while loop
        nextNode = current->next;
        current->next = prevNode;
        prevNode = current;
        current = nextNode;
    }
    head = prevNode;
}
```

Problem #1.4: Suppose we add a new member function called findNthFromLast() to find the N-th node from the end of the list, where N being 1 means the last node, N being 2 the second-to-last, etc. Use the reverse member function to complete the implementation of findNthFromLast() member function. If the list has at least N nodes, then assign to the variable value the number that is stored in that node and return true; otherwise, leave the variable value unchanged and return false. Don't forget to call the reverse function to restore the linked list to its original state.

```
bool LinkedList::findNthFromLast(int N, int &value)
{
    reverse();
    Node *ptr = head;
    int n = 0;
    while(ptr != nullptr) {
        n++;
        if(n == N) {
            value = ptr->num;
            reverse();
            return true;
        }
        ptr = ptr->next;
    }
    reverse();
    return false;
}
```

Problem #1.5: There is a more efficient way to solve problem #1.4 without using reverse() function. The idea is to scan first to count the total number of nodes in the linked list. Once we know how many nodes are in the linked list, the second time we scan the list, the Nth node from the end of list can be easily calculated. Please use this idea to implement this member function findNthFromLast().

```
bool LinkedList::findNthFromLast(int N, int &value)
{
	if( N <= 0 )
		return false;
	Node *ptr = head;
	int i, M = 0;
	// M is the number of nodes in the linked list.
	while(ptr != nullptr)
	{
		M++;
		ptr = ptr->next;
	}
	if( N > M )
		return false;
	for(i=1, ptr = head; i < (M-N+1); i++)
		ptr = ptr->next;
	value = ptr->num;

	return true;
}
// Your implementation should pass the following test program
int main()
{
	int value = 999;
	LinkedList list;
	for(int i=1;i<=10;i++)
		list.addToList(i);
	assert( list.findNthFromLast(0,value) && value == 999);
	assert( ! list.findNthFromLast(11,value) && value == 999);
	assert( list.findNthFromLast(1,value) && value == 10);
	assert( list.findNthFromLast(2,value) && value == 9);
	assert( list.findNthFromLast(10,value) && value == 1);
}
```

Problem #1.6: You and your friends got caught by 100 cannibals in some weird rainforest. Unfortunately, there is no way you can call 911. Fortunately, you overheard these cannibals talking in English about the game they will play on you guys. It's called Josephus Circle that they will make you guys stand in a circle, announce the number N, and every Nth person in the circle will be eaten. Only the last person standing will be freed. Although you are skeptical about whether they are actually cannibals (because they speak English), you decide to write a program to figure out where you should stand in the circle to be the last person standing. Unfortunately, when you open your laptop, you only have a singly linked list C++ program there and a fragmented JosephusCircle() member function.
Hurry up!! Your life is on the line…..

Example: Suppose there are 5 people there in a circle numbered from 1 to 5.
1st round: Number 2 is eaten.
2nd round: Number 4 is eaten.
3rd round: Number 1 is eaten.
4th round: Number 5 is eaten.
Number 3 is the last guy standing…

```
int main()
{
	int i,N;
	for(N=1;N<=7;N++)
	{
		LinkedList a;
		for(i=1;i<=N;i++)
			a.addToList(i);
		cout << "Number " << a.JosephusCircle(2);
		cout << " is the last guy standing." << endl;
	}
}
// Output is:

Number 1 is the last guy standing.
Number 1 is the last guy standing.
Number 3 is the last guy standing.
Number 1 is the last guy standing.
Number 3 is the last guy standing.
Number 5 is the last guy standing.
Number 7 is the last guy standing.
```

```
// You can challenge yourself to write the solution for this problem without looking
// at the partial codes below. See if you can do it!! Should be fun for ya~~~
int LinkedList::JosephusCircle(int N)
{
	Node* ptr = head;
	Node* prev;
	int count;
	while(head->next != nullptr)
	{
		for( count=1; count<N; count++)
		{
			prev = ptr;
			ptr = ptr->next;
			if(ptr == nullptr)
				ptr = head;
		}
		Node *temp = ptr;
		if(head == ptr)
			head = ptr->next;
		else
			prev->next = ptr->next;

		ptr = ptr->next;

		if(ptr == nullptr)
			ptr = head;

		delete temp;		// see you on the other side …
	}
	return head->num;
}
```

Problem #2: Please complete the missing blocks of codes below to make the program generate the following output:

1+2i
6+7i
6+7i
6+7i

```
#include <iostream>
using namespace std;
class Complex {
private:
	double r,i;
public:
	// Please complete the missing codes below.
	Complex(): r(0),i(0) {}
	Complex(int c_r,int c_i): r(c_r),i(c_i) {}
	void output() {
		cout << r << "+" << i << "i" << endl;
	}
};

int main() {
	Complex a,b;

	a = Complex(1,2);	a.output();
	b = Complex(6,7);	b.output();

	a = b;

	a.output();
	b.output();

	return 0;
}
```

Problem #3: Below is a definition of a mystring class and a test main program:

```
#include <iostream>
using namespace std;
// no cstring available for ya....
class mystring
{
public:
	mystring() {    str=new char[1];    str[0] = '\0';    length = 0;    }
	mystring(const char *source);
	mystring(const mystring& s);
	mystring& operator=(const mystring& s);
	~mystring();
	int size() const;                    // return length back;
	char* c_str();                       // return str back
private:
	char *str;
	int length;                          // Can we access length from outside?
};
int main()
{
	int i;
	mystring *s[2];                      // What does this mean? Please explain it.
	for(i=0;i<2;i++)
		s[i] = new mystring();       // What does this mean? Please explain it.

	*s[0] = "Review";                    // What does this mean? Please explain it.
	*s[1] = "Session";
	cout << s[0]->c_str() << " " << s[1]->c_str() << endl;
	// When should we use -> ?.
	*s[0] = "For";
	*s[1] = "CS32";
	cout << s[0]->c_str() << " " << s[1]->c_str() << endl;

	for(i=0;i<2;i++)
		delete s[i];                 // What does this mean? Please explain it.
}
```

Problem #3.1: Please write an implementation for the destructor:

```
mystring::~mystring()
{
	delete [] str;
}
```

Problem #3.2: Please write an implementation for the size() member function. Please make use of the inline keyword.

```
inline int mystring::size() const
{
	return length;
}
```

Problem #3.3: Please write an implementation for the c_str() member function. Please make use of the inline keyword.

```
inline char* mystring::c_str()
{
	return str;
}
```

Problem #3.4: Please write an implementation for the constructor: mystring(const char *source). Please note that you cannot use strlen() and strcpy() to aid your implementation.

```
mystring::mystring(const char *source)
{
	int len = 0;
	const char *ptr = source;

	while (*ptr++) len++;

	length = len;

	str = new char[len+1];
	char *dest = str;

	while( *dest++ = *source++ );
}
```

Problem #3.5: Please write an implementation for the copy constructor: mystring(const mystring& s). Please note that you cannot use strlen() and strcpy() to aid your implementation.

```
mystring::mystring(const mystring& s)
{
	str = new char[s.length+1];
	char *source = s.str;
	char *dest = str;
	while( *dest++ = *source++ );
	length = s.length;
}
```

Problem #3.6: Please write an implementation for the assignment operator: Please note that you cannot use strlen() and strcpy() to aid your implementation.

```
mystring& mystring::operator=(const mystring& s)
{
	if( &s == this)
		return *this;

	delete [] str;
	str = new char[s.length+1];

	char *source = s.str;
	char *dest = str;

	while( *dest++ = *source++ );

	length = s.length;
}
```