Print your name, netID, and lab section day/time neatly in the space provided below; print your name at the upper right corner of every page.

Name: 
NetID: 
Lab Section (Day/Time): 

- This is a **closed book** and **closed notes** exam. In addition, you are not allowed to use any electronic aides of any kind.

- Do all 5 problems in this booklet. Read each question very carefully.

- You should have 7 sheets total (the cover sheet, plus numbered pages 1-12). The last sheet is scratch paper; you may detach it while taking the exam, but must turn it in with the exam when you leave. The page before the scratch paper has the member functions of the **Array** class and the **List** class from the MPs.

- Unless otherwise stated in a problem, assume the best possible design of a particular implementation is being used.

- Unless the problem specifically says otherwise, (1) assume the code compiles, and thus any compiler error is an exam typo (though hopefully there are not any typos), and (2) assume you are NOT allowed to write any helper methods to help solve the problem, nor are you allowed to use additional arrays, lists, or other collection data structures unless we have said you can.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Points</th>
<th>Score</th>
<th>Grader</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>90</strong></td>
<td>**</td>
<td>**</td>
</tr>
</tbody>
</table>
1. **[The Big Three – 15 points].**

Given the following class:

```cpp
// this would be in the .h file
template <class KeyType, class ElemType>
class Dictionary {
    public:
        //default constructor
        Dictionary();

        //copy constructor
        Dictionary(const Dictionary& origVal);

        //destructor
        ~Dictionary();

        // ... plus other functions we don’t care about here
    private:
        Array<KeyType> keys;
        Array<ElemType*> elements;
        int size;
};
```

Assume that all pointers that are in any way part of the implementation of `Dictionary`, get set to either `NULL` or the address of a dynamically object at some point before the object is passed as an argument to the copy constructor. (Or in other words, assume that no pointer you need to read, is pointing to garbage memory.) Write the definition code for the copy constructor for the above class.
(The Big Three, continued)
2. [Analysis – 20 points].

(a) Given the following code, using a \textit{singly-linked} implementation of the List ADT you saw on the MPs (i.e. head and tail pointers, a size variable to hold the size, and no dummy nodes), express (using big-$O$ notation) the order of growth of the worst-case running time of the code below, in terms of $n$. Prove your answer is correct (i.e. explain your answer in enough detail to be convincing). (10 points)

```cpp
List<int> theList;
for (int i = 1; i <= n; i++) // <--- this is the n referred to above
    theList.InsertAfter(i);
theList.Head();
int len = theList.Length();
for (int i = 1; i < len; i++) {
    cout << theList.Retrieve() << endl;
    theList.Remove();
    theList.Head();
}
```
(b) Given the same list implementation as in part (a), express (using big-O notation) the order of growth of the worst-case running time of the code below, in terms of $n$. Prove your answer is correct (i.e. explain your answer in enough detail to be convincing). (10 points)

List<int> theList;
for (int i = 1; i <= n; i++) // <---- this is the n referred to above
    theList.InsertBefore(i);
theList.Tail();
theList.Remove();
3. [Remove Sublist – 20 points].

You have the following ListNode class:

class ListNode {
public:
    int element;
    ListNode* next;
    ListNode* prev;
};

and a doubly-linked list made up of such nodes, with a ListNode pointer head to the first node and with the first node’s prev and the last node’s next equalling NULL. We will assume it is publicly accessible, rather than nested in a class, for this problem.

Write a function RemoveSublist which has three parameters and returns nothing. The first parameter will be a reference to a ListNode pointer. This pointer will point to the head node of a doubly-linked list; we assume this list has at least one node and has no duplicate values.

The second parameter will be an integer that is definitely a value in the list. The third parameter will be a non-negative integer indicating how many nodes in each direction from the second parameter’s node, should be removed. For example, if the parameter list had been 4->502->10->12->7->33->5->821->11->103->90->NULL, and the second and third parameters are 33 and 2, then you want to remove everything within 2 nodes of 33 – that is, 12, 7, 33, 5, and 821. In that case, the resultant list would be 4->502->10->11->103->90->NULL. You can assume that the third parameter will not be so big as to extend past the ends of the list – for example, in the first list above, if 33 is the second parameter, then the third parameter could be 5, but not 6. If the third parameter were 0, only 33 would be removed.

Whatever linked list this results in, the head parameter should be pointing to the first node of that list when you are done.

void RemoveSublist(ListNode*& head, int val, int range) {
    // your code goes here
(RemoveSublist, continued)
4. [Generic Functions – 15 points].

(a) You are given the following generic function:

```cpp
// assumes that the type which iterators point to, supports operator!=
template <class Iter>
int isPalindrome(Iter first, Iter last) {
    if (first == last) // empty range
        return 1;
    else {
        last--; // last--
        while (first != last) {
            if (*first != *last)
                return 0;
            else {
                first++;
                if (first == last)
                    return 1;
                else
                    last--;
            }
        }
        return 1;
    }
}
```

Furthermore, you have a class `list` as seen on the MPs (i.e. with a nested `iterator` class, and you have made the declaration:

```cpp
list<int> theList;
```

and then inserted values such that the list looks as follows (where the asterisk indicates the null position at the end of the list):

```
2 8 3 9 4 0 3 5 7 1 6 *
```

Write some code that uses iterators for the list `theList` that we declared above, and the template function above, to print a 1 if the above list is a palindrome and 0 if it is not. (It is not, but let the template function decide that.) Note that no iterators are declared yet; you will need to do that yourself, if you decide you need iterator variables. (6 points)
(b) Now, we want to change the generic function from part (a) to the following:

```cpp
template <class Iter, class Comparer>
int isPalindrome(Iter first, Iter last, Comparer check) {
    if (first == last) // empty range
        return 1;
    else {
        last--;
        while (first != last) {
            if (!check(*first, *last))
                return 0;
            else {
                first++;
                if (first == last)
                    return 1;
                else
                    last--;
            }
        }
    }
    return 1;
}
```

You want to write a class whose objects can be passed as the third argument to the above function, when the first two arguments above are iterators that point to collections of integers (for example, iterators to lists of integers, or iterators to vectors of integers, or etc.). The class should be such that the \texttt{check(*first, *last)} expression above evaluates to 1 if the two integer arguments are within 10 of each other, and evaluates to 0 otherwise. It is okay to write the definition for this class right into the class declaration itself (i.e. you don't need to divide things up into a .h and .cpp). (9 points)
5. [Stack and Queue Interfaces – 20 points].

Imagine you are given a standard Stack class and Queue class, each of which also has a
no-argument constructor that initializes the data structure to be empty.

You want to write a function Quads which takes as an argument, a reference to a Queue. The
function should break the collection of elements inside the queue into groups of four, and
reverse the groups of four. (You might not have a complete group of four at the end of the
queue.) For example, given the following queue:

```
front    rear
10  -2  0  5  7  2  -8  3  4  14  19  7  3
```

you want to change the queue into the following:

```
front    rear
5  0  -2  10  3  -8  2  7  19  1  14  4  3  7
```

The catch is that we’ve declared a few local integers below for you to use (you don’t have
to use all of them, we’ve just given them to you in case you need them), and a local Stack
and Queue, and you cannot create any other local variables of any kind – not even additional
loop counters or pointers. So, you will need to perform some stack and queue manipulation
to solve this problem.

```cpp
void Quads(Queue<int>& param) {
    int tempVal, i;  // hint : i would be a useful loop counter
    Stack s;  // not the best variable names, but you have
    Queue q;  // less to write this way
    // your code goes here
}```
(Stack and Queue Interfaces, continued)
class Array:
    Array(); // creates array of size 0
    Array(int low, int hi); // creates array with index range (low, hi)
    Array(const Array& origVal); // copy constructor
    ~Array(); // destructor
    const Array& operator=(const Array& origVal); // assignment operator
    Etype& operator[](int index);
    const Etype& operator[](int index) const;
    void Initialize(Etype initElement);
    void SetBounds(int low, int hi); // changes bounds of array
    int Size() const; // returns number of indices in index range
    int Lower() const; // returns lower bound of index range
    int Upper() const; // returns upper bound of index range

class List:
    List(); // creates empty list
    List(const List& origVal); // copy constructor
    ~List(); // destructor
    const List& operator=(const List& origVal); // assignment operator
    void Clear(); // empties an existing list
    void InsertAfter(const Etype& newElem); // inserts after current value
    void InsertBefore(const Etype& newElem); // inserts before current value
    void Remove(); // removes current value
    void Update(const Etype& updateElem); // changes current value to parameter value
    void Head(); // changes current marker to indicate first value
    void Tail(); // changes current marker to indicate last value
    List& operator++(int); // moves current marker one position forward
    List& operator--(int); // moves current marker one position backward
    const Etype& Retrieve() const; // returns the current value
    int Find(const Etype& queryElem); // returns 1 if parameter is in list, else 0
    int Length() const; // returns number of elements in list
    void Print() const; // prints list to screen