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.

<table>
<thead>
<tr>
<th>Name:</th>
<th>SOLUTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>NetID:</td>
<td></td>
</tr>
<tr>
<td>Lab Section (Day/Time):</td>
<td></td>
</tr>
</tbody>
</table>

- 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>Total</td>
<td>90</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. [Assignment Operator – 15 points].

Given the following class:

```cpp
// this would be in the .h file
template <class Etype>
class Map {
private:
    Array<Etype> items;
    Array<String*> labels;
    Etype* primaryValue;
public:
    Map();
    const Map& operator=(const Map& origVal);
    // ...plus other functions we don’t care about here
};
```

Assume that all pointers that are in any way part of the implementation of Map, get set to either NULL or the address of a dynamically object before you call the assignment operator. (Or in other words, assume that when you call the assignment operator, no pointer is pointing to garbage memory.) Write the definition code for the assignment operator (operator=) for the class Map.

SOLUTION ON NEXT PAGE
(Assignment Operator, continued)

    const Map<Etype>& Map<Etype>::operator=(const Map<Etype>& origVal)
    {
        if (this != &origVal)
        {
            for (int i = labels.Lower(); i <= labels.Upper(); i++)
                delete labels[i];
            delete primaryValue;

            items = origVal.items;
            labels = origVal.labels; // will size it properly
            for (int i = labels.Lower(); i <= labels.Upper(); i++)
                labels[i] = new String(*((origVal.labels)[i]));
            primaryValue = new Etype(*(origVal.primaryValue));
        }
        return *this;
    }
2. [Analysis – 20 points].

(a) Given the following code, using a singly-linked implementation of the List ADT you saw on MP3, express (using big-$O$ notation) the order of growth of the 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.Tail();
for (int i = 1; i < theList.Length(); i++) {
    cout << theList.Retrieve() << endl;
    theList--;
}
cout << theList.Retrieve(); // prints first element
return 0;
```

ANSWER: The running time is $O(n^2)$. Inserting each of the $n$ elements will take constant time, and since there are $n$ of those insertions, the first loop is linear time. The call to the Tail() function would either be constant time (if there were a tail pointer as part of the implementation) or else linear time (if there were NOT a tail pointer as part of the implementation and you had to traverse all the way through the list to get to the end). Either way, the running time for the first four lines together is linear time.

The second loop will perform a Retrieve() on every node except the first node, starting at the last node and moving backwards. Each Retrieve() is constant time, but the act of moving backwards one position will be linear time, since you need to start at the beginning and traverse to “the node before the current position” in order to move backwards on a singly-linked list. So you are running a linear-time body of the loop $n-1$ times, and that is overall quadratic time. (If you want to be more precise, the first time theList--; is run, it will require traversing down $n-1$ nodes, the second time it is run requires a traversal down $n-2$ nodes, the third time it is run requires a traversal down $n-3$ nodes, and so on. And the sum $(n-1) + (n-2) + (n-3) + \ldots + 1$ turns out to be a quadratic function of $n$.)

So, since linear plus quadratic is quadratic, the running time is quadratic.
(You would not need to be quite so verbose in your own solution.)
(b) Imagine we have the following array-based implementation of a stack:

```cpp
class Stack {
    private:
        Array<int> theStack;
        int numElements;
        // rest of class, including public functions
    }
```

where `theStack.Size()` gives you the size of the array, which will be indexed from 1 through `theStack.Size()`, `numElements` stores the number of elements in the actual stack itself (could be less than the total amount of space in the array), and the stack is placed in order in the array so that the top element is at `theStack[1]`, rather than at `theStack[numElements]` as in lecture. You have *no other member variables* for this `Stack` class.

Given a stack of size $n$ implemented as above, what is the order of growth of the running time of `Pop()`, in terms of $n$? Express your answer in big-$O$ notation. Prove your answer (i.e. explain your answer in detail sufficient enough to be convincing).

(10 points)

**Answer:** The running time will be $O(n)$. Since there are no additional member variables, you cannot be using a “circular array” to implement this stack. And so the only way to actually get a `Pop()` operation to work, is to shift all the elements of the stack in the index range 2 through `numElements`, to the left by one cell, thus moving that range to the index range 1 through `numElements-1`. Since it takes constant time to shift each value one cell to the left, and we shift $n-1$ values, the overall running time must be linear.

(You would not need to be quite so verbose in your own solution.)
3. [Move Tens – 20 points].

You have the following ListNode class:

```cpp
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 `MoveTens` which has one parameter and returns nothing. The parameter will be a reference to a ListNode pointer. This pointer will point to the head node of a doubly-linked list (and thus would be NULL if the list were empty). This list will hold only positive integers, and will have the prev of the first node and the next of the last node both pointing to NULL.

This function should move every node containing a 2-digit number to the start of the list. All the nodes you move should remain in the same order relative to each other, and all the nodes you do not move should remain in the same order, relative to each other. For example, if the parameter list had been 4->502->10->12->7->33->5->821->11->103->NULL, then you are moving 10, 12, 33, and 11 to the front of the list but keeping them in that order (10, 12, 33, 11). And the values you did not move stay in the same order they were in to begin with. So, after the function has run, the list should be 10->12->33->11->4->502->7->5->821->103->NULL.

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

```cpp
void MoveTens(ListNode*& head) {
    // your code goes here
}
```

SOLUTION ON NEXT PAGE
(Move Tens, continued)

void MoveTens(ListNode*& head) {
    // your code goes here
    ListNode* temp = head;
    ListNode* temp2;
    ListNode* newHead = NULL;
    ListNode* newTail = NULL;
    while (temp != NULL) {
        temp2 = temp->next; // save value *after* the current one
        if ((temp->element >= 10) && (temp->element < 100)) {

            // remove node from original list
            if (temp->prev != NULL) // if there’s a prev node
                temp->prev->next = temp->next; // prev node should point to next node
            else // otherwise, this is first node, so
                head = head->next; // next node is new first node

            if (temp->next != NULL) // if there’s a next node
                temp->next->prev = temp->prev; // next node should point to prev node

            // append node to end of new list
            if (newHead == NULL) { // first node of new list
                newHead = newTail = temp;
                newHead->prev = NULL;
            }
            else { // not first node of new list
                newTail->next = temp;
                temp->prev = newTail;
                newTail = temp;
            }
        }
    }
    temp = temp2; // now make our "saved value" from before, the current value

    if (head != NULL) // if there’s still some of old list left
        head->prev = newTail; // point first node of old list to last node of new

    if (newTail != NULL) { // if there’s anything in the new list
        newTail->next = head; // last node of new list points to beginning of old,
        head = newHead; // and the "head" pointer points to start of new
    }
}
4. [Generic Functions – 15 points].

(a) You are given the following generic function:

```cpp
template <class Iter>
void printEveryOther(Iter first, Iter last) {
    while (first != last) {
        cout << *first << " ";
        first++;
        if (first != last)
            first++;
    }
    cout << "the end!" << endl;
}
```

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 the following line of text. Note that no iterators are declared yet; you will need to do that yourself. (8 points)

```cpp
8 9 0 5 the end!
```

```cpp
list<int>::iterator it1, it2;
it1 = theList.begin();
it1++;
it2 = theList.end();
it2--;
it2--;
printEveryOther(it1, it2);
```
(b) Now, we want to change the generic function from part (a) to the following:

```cpp
template <class Iter, class Comparer>
void VerifyAndPrintEveryOther(Iter first, Iter last, Comparer check) {
    while (first != last) {
        if (check(*first))
            cout << *first << " ";
        first++;
        if (first != last)
            first++;
    }
    cout << "the end!" << endl;
}
```

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 `check(*first)` expression above evaluates to 1 if `*first` is greater than or equal to 5, and returns 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). (7 points)

```cpp
class Foo
{
    int operator()(int value)
    {
        return (value >= 5);
    }
};
```
5. [Stack and Queue Interfaces – 20 points].

Imagine you are given a standard Stack class and Queue class, each of which also has a Size() function that tells you how many items are in the data structure, and a no-argument constructor that initializes the data structure to be empty.

You want to write a function Thirds which takes as an argument, a reference to a Queue. The function should break the collection of elements inside the queue into three equal-sized pieces (If the number of elements is not a multiple of three, then the piece closest to the front gets an extra value and, if there is an additional extra value, the middle section would get that one.) The Queue should be changed so that the second section of the Queue is reversed, and the first and third sections are swapped. For example, given the following queue:

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

you want to change the queue into the following:

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

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 the only other local variables you can create and use are new Queues and new Stacks.

```c
void Thirds(Queue<int>& param) {
    int temp1, temp2, temp3;
    // your code goes here
}
```

SOLUTION ON NEXT PAGE
(Stack and Queue Interfaces, continued)

```cpp
void Thirds(Queue<int>& param) {
    int temp1, temp2, temp3;
    // your code goes here

    temp1 = param.Size()/3;
    if (param.Size() % 3 != 0)
        temp1++; // size of front section
    temp2 = param.Size()/3;
    if (param.Size() % 3 == 2)
        temp2++; // size of middle section

    Queue<int> frontHolder;
    Stack<int> reverser;
    for (temp3 = 1; temp3 <= temp1; temp3++)
        frontHolder.Enqueue(param.Dequeue());

    for (temp3 = 1; temp3 <= temp2; temp3++)
        reverser.Push(param.Dequeue());

    for (temp3 = 1; temp3 <= temp2; temp3++)
        param.Enqueue(reverser.Pop());

    for (temp3 = 1; temp3 <= temp1; temp3++)
        param.Enqueue(frontHolder.Dequeue());
}
```
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
    const Etype& operator[](int index) const;
    Etype& operator[](int index);
    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