Overview

- Test 3 will be held Friday, November 18, 2005, 20:00-3:50pm, Darrin 308. No make-ups will be given except as pre-arranged by written excuse from the Dean of Students office.

- Coverage: the emphasis is the material covered in Lectures 13-19, Labs 7-11, and HW 6. Material from earlier in the semester may be covered as well.

- Closed-book and closed-notes. Photocopies of the “Details” sections from the text will be provided.

- Below are sample questions. Solutions will be posted on-line.

- How to study?
  - Review the lecture notes
  - Review and re-do lecture exercises, lab and homework problems.
  - Do the practice problems. Practice writing solutions using pencil (or pen) and paper.

- The material on linked lists from Lecture 19 and Lab 11 will be covered on the test, but the material from Lecture 20 on the list class will not.

Practice Problems

1. You are given a map that associates strings with lists of strings. The definition is

```cpp
map<string, list<string> > words;
```

Write a function that counts the number of key strings that are in their own associated list. For example, suppose the map contained just the following three key strings and lists

<table>
<thead>
<tr>
<th>string</th>
<th>list</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>car, cab, jet, apple</td>
</tr>
<tr>
<td>car</td>
<td>horse, car, train, car</td>
</tr>
<tr>
<td>jet</td>
<td>buggy, abc</td>
</tr>
</tbody>
</table>

Then the function should return the value 1, since only car is in its own list. Start from the function prototype

```cpp
int count( map<string, list<string> > const& words )
```
2. Write a constructor for the Vec<T> class that builds a Vec from a std::list. The form of the constructor is shown in the following excerpt from the class declaration:

```cpp
template <class T>
class Vec {
public:
    Vec( list<T> const& s );
    // ... lots of other declarations here
private:
    T* m_data; // Pointer to first location in the allocated array
    size_type m_size; // Number of elements stored in the vector
    size_type m_alloc; // Number of array locations allocated. m_size <= m_alloc at all times
};
```

You may not use any member functions of Vec<T>. Here is an example of how the constructor should work.

```cpp
list<int> u;
u.push_back(5); u.push_back(-2); u.push_back(14);
Vec<int> v(u);
// u now has size 3 and contains the integers 5, -2 and 14 in that order.
```

3. You are given two maps of type

```cpp
typedef map< string, list<string> > CoursesType;
```

Think of these as representing two maps associating course ids with lists of student ids. Your problem is to write a function to merge two CoursesType maps into a single map. When a course id is in both maps, the two lists associated with it, one from each map, must be merged into a single list. A pair associating the course id and the merged student id list should be placed in the final map. When a course id is in only one of the maps, the course id / student id list pair should be copied into the final map unchanged. You may assume that the following function has already been written and works correctly:

```cpp
list<string> merge( list<string> const& a, list<string> const& b );
```
4. Given an array of integers, `intarray`, and a number of array elements, `n`, write a short code segment that uses **pointer arithmetic and dereferencing** to add every second entry in the array. For example, when `intarray` is

```
0 1  2  3  4  5  6  7  8
1 16 -3  2 76  9  3  6
```

and `n` is 9, the segment should add `1 + 4 + 2 + 9 + 6` to get 22. Store the result in a variable called `sum`.

5. Show the output from the following code segment.

```cpp
int x = 45;
int y = 30;
int *p = &x;
p = 20;
cout << "a: x = " << x << endl;

int *q = &y;
int temp = *p;
p = *q;
*q = temp;
cout << "b: x = " << x << " y = " << y << endl;

int * r = p;
p = q;
q = r;
cout << "c: *p = " << *p << " *q = " << *q << endl;
cout << "d: x = " << x << " y = " << y << endl;
```

6. Write a `Vec<T>` class member function that creates a new `Vec<T>` from the current `Vec<T>` that stores the same values as the original vector but in reverse order. The function prototype is

```
template <class T>
Vec<T> Vec<T>::reverse() const;
```

Recall that `Vec<T>` class objects have three member variables: Recall that `Vec<T>` class objects have three member variables:

```
T* m_data; // Pointer to first location in the allocated array
size_type m_size; // Number of elements stored in the vector
size_type m_alloc; // Number of array locations allocated. m_size <= m_alloc at all times
```
7. (This problem may be a bit harder than what will be on the exam because we haven’t emphasized passing pointers by reference, but it should still be a good practice problem.) Write a function that takes an array of floating point numbers and copies its values into two new arrays that must be allocated in the function, one containing only the negative numbers from the original array, and the other containing the non-negative numbers from the original array. For example, if the original array is

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>-1.3</td>
<td>5.2</td>
<td>8.7</td>
<td>0.0</td>
<td>-4.5</td>
<td>7.8</td>
<td>-9.1</td>
<td>3.5</td>
<td>6.6</td>
</tr>
</tbody>
</table>

Then the resulting array containing the negative values would be

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>-1.3</td>
<td>-4.5</td>
<td>-9.1</td>
</tr>
</tbody>
</table>

and the resulting array containing the non-negative values would be

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5.2</td>
<td>8.7</td>
<td>0.0</td>
<td>7.8</td>
<td>3.5</td>
<td>6.6</td>
</tr>
</tbody>
</table>

(a) Start by writing the function prototype. Think about what parameters (6 of them) you need, what their types should be, and how they should be passed.

(b) Now write the code of the actual function. You do not need to write the prototype over again. Do not allocate any more space for the new arrays than is necessary.

(c) Compare this to a version that is based on vectors or lists.

8. What is the output of the following code?

```c++
int * a = new int[4];

cout << "A: ";
for( unsigned int i=0; i<4; ++i ) cout << a[i] << " ";
cout << endl;

for( int * b = a; b != a+4; b += 2 ) *b = b-a;

for( unsigned int i=0; i<4; ++i ) cout << a[i] << " ";
cout << endl;

int * c = a;
c[3] = 14;
c[1] = -2;

cout << "C: ";
for( unsigned int i=0; i<4; ++i ) cout << a[i] << " ";
cout << endl;
```
9. Consider the public stack interface.

```cpp
template <class T>
class stack {
public:
    stack();
    stack( stack<T> const& other );
    ~stack();
    void push( T const& value );
    void pop( );
    T const& top( ) const;
    int size();
    bool empty();
};
```

In Lab 10 you implemented the stack using a `std::vector`. In this question you are not allowed to use either a `std::vector` or a `std::list`. Instead you are to implement the stack using a dynamically-allocated array. To answer the question, show what private member variables are needed and provide the implementation of the default constructor, the destructor, `stack<T>::push`, and `stack<T>::pop`. All operations should be as efficient as possible.
10. Write a function to create a new singly-linked list that is a COPY of a sublist of an existing list. The prototype is

```cpp
template <class T>
Node<T>* Sublist( Node<T>* head, int low, int high )
```

The `Node` class is:

```cpp
template <class T>
class Node {
    public:
        T value;
        Node* next;
    };
```

The new list will contain `high-low+1` nodes, which are copies of the values in the nodes occupying positions `low` up through and including `high` of the list pointed to by `head`. The function should return the pointer to the first node in the new list. For example, in the following drawing the original list is shown on top and the new list created by the function when `low==2` and `high==4` is shown below.

**Original list**

```
head -> 3.1 -> 2.4 -> 8.7 -> 9.4 -> 14.2 -> 0.9 -> 0
```

**New list**

```
nhead -> 2.4 -> 8.7 -> 9.4
```

A pointer to the first node of this new list should be returned. (In the drawing this would be the value of `nhead`.) You may assume the original list contains at least `low` nodes. If it contains fewer than `high` nodes, then stop copying at the end of the original list.