CSCI-1200 Computer Science II — Spring 2006
Test 3 — Practice Problem Solutions

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

```
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:

string	list
abc	car, cab, jet, apple
car	horse, car, train, car
jet	buggy, abc

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

```
int count(map<string, list<string> > const& words)
```

Solution:

```
int count(map<string, list<string> > const& words) {
  int num = 0;
  for (map<string,list<string> >::const_iterator p = words.begin(); p != words.end(); ++p) {
    const string& key = p->first;
    for (list<string>::const_iterator q = p->second.begin();
         q != p->second.end() && key != *q; ++q)
      ;  // empty loop body
    if (q != p->second.end()) num ++;
  }
  return num;
}
```

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:

```
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 always <= m_alloc
};
```

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

```
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.
```

Start from:

```
template <class T>
Vec<T>::Vec(list<T> const& s)
```

Solution: This solution uses pointers instead of array indexing.

```
template <class T> Vec<T>::Vec(list<T> const& s) {
  m_alloc = m_size = s.size();
  m_data = new T[m_size];
  T * p = m_data;
  for (list<T>::const_iterator itr = s.begin(); itr != s.end(); ++itr, ++p)
    *p = *itr;
}
```

3. You are given two maps of type:

```
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:

```
list<string> merge(list<string> const& a, list<string> const& b);
```

Solution: The first solution uses a modification of the idea of merging two sorted lists. It is based on the understanding that maps are sorted. Once this is done, the only tricky parts are dealing with the pairs and merging the lists for common keys.

```
CoursesType merge(CoursesType const& m1, CoursesType const& m2) {
  CoursesType result;
  CoursesType::const_iterator p = m1.begin(), q = m2.begin();

  while (p != m1.end() && q != m2.end()) {
    if (p->first < q->first) {
      result.insert(*p);
      ++ p;
    } else if (q->first < p->first) {
      result.insert(*q);
      ++q;
    } else {
      result.insert(make_pair(p->first, merge(p->second,q->second)));
      ++p; ++q;
    }
  }

  for(; p != m1.end(); ++p) result.insert(*p);
  for(; q != m2.end(); ++q) result.insert(*q);
  return result;
}
```

Another version (shorter but slightly slower):

```
CoursesType merge(CoursesType const& m1, CoursesType const& m2) {
  CoursesType result(m1);    // copy m1 into the result

  // Step through m2 in order
  for (CoursesType::const_iterator q = m2.begin(); q != m2.end(); ++q) {
    // Look for the m2 key in m1.
    CoursesType::const_iterator p = m1.find(q->first);

    // If it is not there, insert the pair from m2 into the result.
    if (p == m1.end()) {
      result.insert(*q);
    }

    // Otherwise, merge the two lists associated with the key and
    // change the value in the map.
    else {
      result[ p->first ] = merge(p->second, q->second);
    }
  }

  return result;
}
```

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	4	-3	2	76	9	3	6

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

Solution:

```
sum = 0;
for (int *p = intarray; p < intarray+n; p+=2)
  sum += *p;
```

5. Show the output from the following code segment:

```
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;
```

Solution:

```
a:  x = 20
b:  x = 30, y = 20
c:  *p = 20, *q = 30
d:  x = 30, y = 20
```

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:

```
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 always <= m_alloc
```

Solution: The confusing part of the problem is that the new vector being created is not the current object — the one referred to by the `this` pointer. Instead it is created as a local variable. Still, since we are inside the `Vec` class, we have direct access to its member variables. The solution uses pointers, but subscripting would work just as well.

```
template <class T>
Vec<T> Vec<T>::reverse() const {
  Vec<T> new_vec;
  new_vec.m_size = new_vec.m_alloc = this->m_size;
  new_vec.m_data = new T[new_vec.m_size];
  for (T * p = new_vec.m_data, *q = this->m_data+this->m_size-1;
       q >= this->m_data; p++, q--)
    *p = *q;
  return new_vec;
}
```

7. 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:

0	1	2	3	4	5	6	7	8
-1.3	5.2	8.7	0.0	-4.5	7.8	-9.1	3.5	6.6

Then the resulting array containing the negative values would be:

0	1	2
-1.3	-4.5	-9.1

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

0	1	2	3	4	5
5.2	8.7	0.0	7.8	3.5	6.6

(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.

Solution: Here is the whole thing, including the prototype. Note that the pointers must be passed by reference and a separate count variable must be passed by reference for each of the arrays. See below for a simpler solution based on vectors.

```
void split(float floatarr[], int n,
   float*& negatives, int& neg_count, float*& positives, int& pos_count) {
  int i;
  neg_count = 0;
  for (i=0; i<n; ++i)
    if (floatarr[ i ] < 0)
      neg_count ++ ;
  pos_count = n - neg_count;

  negatives = new float[ neg_count ];
  positives = new float[ pos_count ];
  int ni = 0, pi = 0;
  for (i=0; i<n; ++i)
    if (floatarr[ i ] < 0) {
      negatives[ ni ] = floatarr[ i ];
      ++ ni;
    } else {
      positives[ pi ] = floatarr[ i ];
      pi ++ ;
    }
}
```

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

Solution: A solution based on vectors (or lists) will create two vectors, `negatives` and `positives` and use the `push_back` member function to store the floats in the appropriate vector.

```
void split(const vector<float>& in_floats,
   vector<float>& negatives,
   vector<float>& positives) {
  negatives.clear();
  positives.clear();
  for (unsigned int i=0; i<in_floats.size(); ++i)
    if (in_floats[i] < 0)
      negatives.push_back(in_floats[i]);
    else
      positives.push_back(in_floats[i]);
}
```

8. What is the output of the following code?

```
int * a = new int[4];
a[0] = 5; a[1] = 10; a[2] = 15; a[3] = 20;
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;
cout << "B: ";
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;
```

Solution:

```
A: 5 10 15 20
B: 0 10 2 20
C: 0 -2 2 14
```

9. Stacks and queues are very simple sequence containers in which items are only added and removed from the end. In a stack, all work is done on just one end, called the `top`. Hence, when an item is removed, it will be the item most recently added. As a result, a stack is called a LIFO structure, for "Last In First Out". In a queue, items are added to the end, usually called the `rear` or `back`, and removed from the other end, usually called the `front`. Hence, when an item is removed it will be the item that has been in the queue longer than any other item currently in the queue. As a result, a queue is called a FIFO structure, for "First In First Out". A fundamental property distinguishing stacks and queues from other containers is that **items in the middle of the sequence may not be accessed or removed.** One effect of this is that neither stacks nor queues have iterators.

   Stacks and queues are implemented in the standard library as templated containers. The include files are just called `stack` and `queue`, as in:

```
#include <stack>
#include <queue>
```

   The definition of stack and queue objects is pretty much what you might guess, e.g.:

```
std::stack<int> s;
std::queue<char> q;
```

   You can read more about these STL data structures from any reference, e.g.:

```
http://www.sgi.com/tech/stl/stack.html
http://www.sgi.com/tech/stl/queue.html
```

   (a) Write a program that uses a stack and a queue to determine if the alphabetic characters in a line of input form a palindrome. The program must read in the characters from `cin` one at a time until encountering the `'\n'` char (use `cin.get(c)` rather than `cin >> c`), convert all alphabetic chars to lower case, and store the letters in a stack or a queue or both. After the line of input has been read, the program must empty the container(s) used, determine if the line is a palindrome while doing so, and output an appropriate message.

   Solution:

```
int main() {
  queue<char> q;
  stack<char> s;
  char c;

  while (cin.get(c) && c != '\n') {
    if (isalpha(c)) {
      c = tolower(c);
      q.push(c);
      s.push(c);
    }
  }

  bool is_palindrome = true;
  while (is_palindrome && !q.empty()) {
    if (q.front() != s.top())
      is_palindrome = false;
    else {
      q.pop(); s.pop();
    }
  }

  if (is_palindrome)
    cout << "Palindrome." << endl;
```

```
  else
    cout << "NOT palindrome" << endl;
  return 0;
}
```

(b) In this part we'll re-implement the stack data structure from scratch. You are not allowed to use either `std::vector` or `std::list`. Instead you will use a dynamically-allocated array. Here's the begining of the class definition:

```
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();
};
```

To answer this 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.

Solution:

```
template <class T>
class stack {
public:
  stack() : arr(0), size(0), alloc_size(0) { }
  stack(stack<T> const& other);
  ~stack() { delete [] arr; }

  void push(T const& value) {
    // this function is a bit long to appear in a header file
    // but we chose to inline the function for efficiency
    if (size == alloc_size) {
      int alloc_size *= 2;
      if (alloc_size < 2) alloc_size = 2;  // could be 1 or 3 or ...
      T * new_arr = new T[alloc_size];
      for (int i=0; i<size; ++i)  new_arr[i] = arr[i];
      delete [] arr;
      arr = new_arr;
    }
    arr[size] = value;
    ++ size;
  }

  void pop() {-- size;}
  T const& top() const;
  int size();
  bool empty();
private:
  T * arr;
  int size;
  int alloc_size;
};
```