CSCI-1200 Computer Science II — Spring 2006
Test 2 — Practice Problem Solutions

1. Write a code segment that copies the contents of a string into a list of char in reverse order.

   Solution:

```
//  assume the string is s and s has been initialized
list<char> result;
for (int i=0; i<s.size(); ++i)
  result.push_front(s[i]);
```

2. Write a code segment that removes all occurrences of the letter 'c' from a string. Consider both uppercase 'C' and lowercase 'c'. For example, the string "`Chocolate`" would become the string "`hoolate`".

   Solution: There are many possible solutions to this problem. Here is one that requires only O(N) time (where N is the size of the string) by delaying the copying.

```
// assume the string is s
unsigned int i=0, j=0;
while (j != s.size()) {
  if (s[j] != 'C' && s[j] != 'c')        {
    S[i] = s[j]; //  copy from location j to i
    ++i;  ++j;   //  increment both
  } else
    ++j;         //  increment just j, thereby skipping the 'C' or 'c'
}
int old_size = s.size();  // Remember this because pop_back changes the size
for (; i<old_size; ++i) s.pop_back();
//  The previous two lines could also be accomplished with just
//  the following line (which is commented out).
//  s.erase(s.begin()+i, s.end()); // remove everything from i to the end
```

3. Write a function that rearranges a list of doubles so that all the negative values come before all the non-negative values AND the order of the negative values is preserved AND the order of the positive values is preserved. For example, if the list contains:

```
-1.3, 5.2, 8.7, 0.0, -4.5, 7.8, -9.1, 3.5, 6.6
```

   Then the modified list should contain:

```
-1.3, -4.5, -9.1, 5.2, 8.7, 0.0, 7.8, 3.5, 6.6
```

   This is a challenging problem, but it is good practice. Try to do it in two different ways: one without using an extra list and one using an extra list.

   Solution: Here's an "in-place" solution which does not use an extra list. Whenever a non-negative number is seen it is removed and placed on the back of the list. Negative numbers are skipped. The trick to ensuring this works is using a separate counter to ensure that all items in the list are tested exactly once. Otherwise, some will be examined multiple times.

```
void rearrange(list<double>& dbl) {
  unsigned int sz = dbl.size();
  unsigned int i;
  list<double>::iterator itr = dbl.begin();
  for (i=0; i<sz; ++i) {
    if (*itr < 0)
      ++itr;
```

```
    else {
      dbl.push_back(*itr);
      itr = dbl.erase(itr);
    }
  }
}
```

Here's a second solution which uses an extra list and two passes through the list. The first pass puts the negative numbers on the front and the second pass puts the positive numbers on the back.

```
void rearrange(list<double>& dbl) {
  list<double> temp;
  list<double>::iterator i;
  for (i = dbl.begin(); i != dbl.end(); ++i)
    if (*i < 0)
      temp.push_back(*i);
  for (i = dbl.begin(); i != dbl.end(); ++i)
    if (*i >= 0)
      temp.push_back(*i);
  dbl = temp;
}
```

4. Write a function that determines if the letters in a string are in alphabetical order. Whitespace characters and punctuation characters in the string should be ignored in deciding if the letters are in alphabetical order. For example:

   ```
   b *((* B  Eee& &^E fg rz!!
   ```

   is in alphabetical order, but:

   ```
   b *((* B  Eee& &^Ea fg rz!!
   ```

   is not.

   Solution:

   ```
   bool is_alphabetical(string const& s) {
     char prev_letter = 'a';
     for (unsigned int i=0; i<s.size(); ++i) {
       if (isalpha(s[i])) {
         char letter = tolower(s[i]);
         if (prev_letter > letter) return false;
         prev_letter = letter;
       }
     }
     return true;
   }
   ```

5. Consider the following start to the declaration of a `Course` class.

   ```
   class Course {
   public:
     Course(const string& id, unsigned int max_stu)
        : course_id(id), max_students(max_stu) {}



   private:
     string course_id;
     list<string> students;
     unsigned int max_students;
   };
   ```

Use this in solving each of the following problems.

(a) Provide three member functions: one returns the maximum number of students allowed in the `Course`, a second returns the number of students enrolled, and a third returns a `bool` indicating whether or not any openings remain in the `Course`. Provide both the prototype in the declaration above and the member function implementation.

Solution: Inside the class declaration, they can add:

```
int max_students_allowed() const { return max_students; }
int students_enrolled() const { return students.size(); }
int remaining_spots() const { return max_students - students.size();   }
```

They can also do this with class declarations inside and the function bodies outside, as in:

```
int Course :: max_students_allowed() const { return max_students; }
```

(b) Write a function that sorts course objects by increasing enrollment. In other words, the course having the fewest students should be first. If two courses have the same number of students, the course with the smaller maximum number of students allowed should be earlier in the sorted vector.

Solution:

```
bool compare_enroll(const Course& c1, const Course& c2) {
  return c1.students_enrolled() < c2.students_enrolled() ||
    (c1.students_enrolled() == c2.students_enrolled() &&
     c1.max_students_allowed() < c2.max_students_allowed());
}

void order_by_enrollemnt(vector<Course>& classes) {
  sort(classes.begin(), classes.end(), compare_enroll);
}
```

(c) Write a member function of `Course` called `merge`. This function should take another `Course` object as an argument. All students should be removed from the passed `Course` and placed in the current course (the one on which the function is called). You may assume (just for this problem) that no students are enrolled in both courses and there is enough room in the current course.

As an example if `cs2` is of type `Course` and has 15 students and `baskets` is of type `Course` and has 5 students, then after the call `cs2.merge(baskets);` `baskets` will have no students and `cs2` will have 20.

Solution: Here are two solutions. The first is based on iterating through the list and then clearing. The second is based on popping and pushing. These need to be added to the declaration, but don't worry about it.

```
void Course :: merge(Course& other) {
  for (list<string>::iterator p = other.students.begin();
    p != other.students.end(); ++ p)
  students.push_back(*p);
  other.students.clear();
}

void Course ::  merge2(Course& other) {
  while (!other.students.empty()) {
    students.push_back(other.students.front());
    other.students.pop_back();
  }
}
```

6. Write a function that takes a vector of strings as an argument and returns the number of vowels that appear in the string. A vowel is defined as an 'a', 'e', 'i', 'o' or 'u'. For example, if the vector contains the strings:

```
abe  lincoln  went  to  the  white  house
```

Your function should return the value 12. You may assume that all letters are lower case. Here is the function prototype:

```
int count_vowels(const vector<string>& strings)
```

Solution:

```
int count_vowels(const vector<string>& strings) {
  int count = 0;
  for (unsigned int i=0; i<strings.size(); ++i)
    for (unsigned int j=0; j<strings[i].size(); ++j)
      if (strings[i][j] == 'a' || strings[i][j] = 'e' ||
          strings[i][j] == 'i' || strings[i][j] = 'o' ||
          strings[i][j] == 'u')
        ++count;
  return count;
}
```

7. Write a function that takes a list of doubles and copies its values into two lists of doubles, one containing only the negative numbers from the original list, the other containing only the positive numbers. Values that are 0 should not be in either list. For example, if the original list contains the values:

```
-1.3, 5.2, 8.7, -4.5, 0.0, 7.8, -9.1, 3.5, 6.6
```

then the resulting list of negative numbers should contain:

```
-1.3, -4.5, -9.1
```

and the resulting list of positive numbers should contain:

```
5.2, 8.7, 7.8, 3.5, 6.6
```

Start this problem by writing the function prototype as you think it should appear and then write the code.

Solution:

```
void copy_pos_neg(const list<double>& original,
                  list<double>& negatives,  // passing by reference is required
                  list<double>& positives) { // passing by reference is required
  negatives.clear();  positives.clear();
  for (list<double>::const_iterator i = original.begin();
       i != original.end(); ++ i) {
    if (*i < 0)
      negatives.push_back(*i);
    else if (*i > 0)
      positives.push_back(*i);
  }
}
```

8. Write a program that reads a sequence of strings from `cin` (up to the end of file) and determines the fraction of strings for which the vowels appear in alphabetical order. For example if the strings are:

```
Aardvark  birthday anybody count  car     peppermint
Ohio!! enough  re-iterate baseball
```

Then the output should be: 0.6 because there are 10 strings and 6 (`Aardvark`, `anybody`, `count`, `car`, `peppermint` and `enough`) contain their vowels in alphabetical order. You do not need to write down any `#include` statements, and you are welcome to write additional functions.

Solution: Here is a function to determine whether or not the vowels are in order:

```
bool vowels_in_order(string const& s) {
  char last_vowel = 'a';
  for (unsigned int i=0; i<s.size(); ++i) {
    char c = tolower(s[i]);
    if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') {
      if (last_vowel > c) return false;
    last_vowel = c;
  }
  return true;
}
```

Here is the main program which uses this function:

```
int main() {
  string s;
  int all_count = 0, in_order = 0;
  while (cin >> s) {
    if (vowels_in_order(s)) in_order ++;
    all_count ++;
  }
  cout << in_order / (float) all_count < endl;
  return 0;
}
```

9. Write a function that merges two lists of float, each of which already in increasing order, to form a single list of floats. Try do this in two ways. In the first way, copy the values into a w vector. In the second, insert the values from one list into the her. Use the `list<T>::insert` function.

Solution:

```
void merge1(list<float> const& a, list<float> const& b, list<float> &c) {
  c.clear();
  list<float>::const_iterator p = a.begin();
  list<float>::const_iterator q = b.begin();
  while (p != a.end() && q != b.end()) {
    if (*p < *q) {
      c.push_back(*p);
      ++p;
    } else {
      c.push_back(*q);
      ++q;
    }
  }
  for (; p != a.end(); ++p) c.push_back(*p);
  for (; q != b.end(); ++q) c.push_back(*q);
}

void merge2(list<float> &a, list<float> &b) {
  list<float>::iterator p = a.begin();
  list<float>::iterator q = b.begin();
  while (p != a.end() && q != b.end()) {
    if (*p < *q)
      ++p;
    else {
```

```
      a.insert(p, *q);
      ++ q;
    }
  }
  for (; q != b.end(); ++q) a.push_back(*q);
}
```

10. Write a function that finds the mode of a vector of integers. This is the integer that occurs most often. If two (or more) integers occur the same number of times then you should return the smallest. More computationally efficient solutions are better. For example, if the vector contains the values:

    ```
    15, 45, 32, 16, 32, 83, 45, 89, 32, 15, 83
    ```

    the function should return the value 32 because it occurs 3 times and no other value occurs more than twice.

    Solution: The solution below is fairly simple. After sorting, every time the value does not change (it is equal to the previous value), the count is incremented AND a test is made to determine if the value should become the mode. The latter test could be made slightly more efficient by only doing the test when the value isn't the same as the previous one, but this makes the code have a few more special case checks.

```
int find_mode(vector<int> const& values) {
  vector<int> copy = values;
  sort(copy.begin(), copy.end());
  int mode = copy[0];
  int mode_count = 1;
  int current_count = 1;
  for (unsigned int i=1; i<copy.size(); ++i) {
    if (copy[i] == copy[i-1]) {
      current_count ++ ;
      if (current_count > mode_count) {
        mode = copy[i];
        mode_count = current_count;
      }
    } else {
      current_count = 1;
    }
  }
}
```

11. Write a **recursive** function to add two non-negative integers using only a limited set of operations: adding 1 to a value, subtracting 1 from a value, and comparing a value to 0. No loops are allowed in the function. The function prototype should be:

    ```
    int Add(int m, int n)
    ```

    Solution:

```
int Add(int m, int n) {
  if (n == 0)  return m;
  else  return 1 + Add(m, n-1);
}
```

12. Write a **recursive** function to multiply two non-negative integers using only addition, subtraction and comparison operations. No loops are allowed in the function. The function prototype should be:

    ```
    int Multiply(int m, int n)
    ```

    Solution:

```
int Multiply(int m, int n) {
  if (n == 0)  return 0;
  else  return m + Multiply(m, n-1);
}
```