Computer Science II — CSci 1200
Test 2 — Overview and Practice

Overview

- Test 2 will be held **Friday, March 21, 2008 2:00-3:45pm, Darrin 308**. No make-ups will be given except for emergency situations, and even then a written excuse from the Dean of Students office will be required.
- Coverage: Lectures 8-14, Labs 5-8, HW 4-6. Material from earlier in the semester, especially on pointers may also be covered on the test. The coverage from Lecture 14 will not be heavy, since the only practice students will have on this material is Lab 8 and this sheet. The material from Lab 8, Checkpoint 1 on generic functions will not be covered on the test.
- The test will be closed-book and closed-notes. We will provide a single sheet, attached to the test, summarizing some of the syntax from `std::map`. This sheet is at the end of this handout.
- The test questions will be drawn from these questions, often with minor modifications, **and** from the lecture, homework and lab problems. Solutions to some of these questions will be provided a day or two before the exam.
- *How to study?*
  - Work through the sample questions, writing out solutions! You are welcome (and encouraged) to do this with other students.
  - Review and re-do lecture exercises, lab and homework problems.
  - Identify the problems that cause you difficulty and review lecture notes and background reading on these topic areas.

Questions

1. Write a code segment that copies the contents of a string into a list of char in reverse order.
2. Recall that each container class has its own associated `erase` function, which accepts an iterator as an argument, erases the contents of the container referenced by the iterator, and returns a new iterator that references the next location in the container. Is `erase` more efficient for `std::list` or `std::vector` or is the efficiency the same? Briefly justify your answer.
3. 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
   ```

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.

5. Write a function that finds the start and length of the longest sequence of alphanumeric characters in a string. Break ties by returning the start of the first such sequence. If there are no alphanumeric characters then the value of the start can be anything, but the length should be 0. Use the prototype

```
void alphanumeric_sequence( const string & s, int & start, int & length )
```

As an example, if the string `"we6*&6 6234-73ab"` then the function should assign `start==7` and `length==4`.

6. 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;        // currently enrolled students
  unsigned int max_students;    // upper bound on enrollment
};
```

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.

(b) Write a function that sorts a vector of `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.

(c) Write a member function of `Course` called `combine`. 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.

7. Write a function that removes all non-alphabetic characters from a string. For example, if the initial string is

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

then the result string should be

```
"bBEeeEafgrz"
```

8. Write a function that determines whether or not the values in a list are in increasing order. The function prototype is

```
template <class T>
bool is_increasing( list<T> const& t )
```

You must assume that `operator<` is defined for type `T`.

9. Write a function named `filter` that takes in a single argument, `words`, that is a list of strings that are lowercase three-letter words. Your function will remove words from the list as necessary such that in the end, no words have any common characters. For example, given the input sequence of strings:

```
cat bat dog too use zoo won you ace zip bin leg
```

The words “bat”, “too”, and “ace” are removed because they have a common character with “cat”. The words “zoo”, “won”, “you”, and “leg” are removed because they have a common character with “dog”. And the word “bin” is removed because it has a common character with “zip”. Then after calling `filter` the variable `words` contains these strings:

```
cat dog use zip
```

10. Write a function called `in_place_merge` that takes two lists, `s` and `t`, both assumed to be sorted into increasing order. At the end of the function, `t` should have all of its original values, plus all of the values from `s` that are not in `t`, and `t` should *still be in increasing order.* For example, if `s` originally contains the values

```
-5, 3, 4, 11, 13, 15, 43
```

and `t` originally contains the values

```
3, 5, 13, 15, 123, 128
```

then after the function call, `t`, should contain the values

```
-5, 3, 4, 5, 11, 13, 15, 43, 123, 128
```

It must do this as efficiently as possible. Here is the function template:

```
template <class T>
void in_place_merge( list<T> const& s, list<T> & t )
```

11. 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
```

Solve it in four different ways:

(a) Assume the values are in a `std::list<double>`. Rearrange the values using an extra list as "scratch" space.

(b) Assume the values are in a `std::list<double>`. Rearrange the values *without* using an extra list (or any other extra container).

(c) Assume the values are stored in a singly-linked list. Rearrange the values *without* using any `new` or `delete` operations.

(d) Assume the values are stored in a doubly-linked list. Rearrange the values *without* using any `new` or `delete` operations.

For the latter two subproblems, you may assume the lists have a dummy head node if you wish.

Note: For this problem and for all remaining problems, you may assume the nodes in a singly-linked list are declared as

```
template <class T>
class Node {
public:
  Node( ) : next(0) {}
  Node( const T& v ) : value(v), next(0) {}
  T value;
  Node<T>* next;
};
```

and the nodes in a doubly-linked list are declared as

```
template <class T>
class Node {
public:
  Node( ) : next(0), prev(0) {}
  Node( const T& v ) : value(v), next(0), prev(0) {}
  T value;
  Node<T>* next;
  Node<T>* prev;
};
```

12. Write a function that takes a pointer to the head of a singly-linked list and returns a pointer to the head of a new linked list that is a *copy of the linked list, with the values in reverse order*. You **may not** assume the existence of a `reverse` function. Here is the prototype

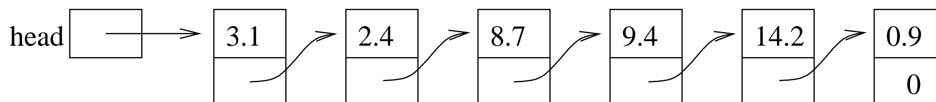
```
template <class T>
Node<T>* reverse_copy( Node<T>* head )
```

13. Write a function to create a new singly-linked list that is a COPY of a sublist of an existing list. The prototype is

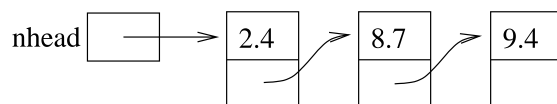
```
template <class T>
Node<T>* sublist( Node<T>* head, int low, int high )
```

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



New list



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.

14. Recall that the `cs2list<T>` class is implemented as a doubly-linked list. Here is what you need from the class declaration, plus two additional function declarations.

```
template <class T>
class cs2list {
private:
  Node<T> * head, * tail;
  int size;
public:
  cs2list() : head(0), tail(0), size(0) {}
  void erase_all_x( const T& x );
  void splice( int k, cs2list<T> & other );
};
```

(a) Implement `erase_all_x`, which should remove all nodes from the `cs2list<T>` that have the value of the parameter `x`. You may make no assumptions other than that `cslist<T>` is in a valid state (i.e. no incorrectly assigned pointers, etc.), and **you may not use any member functions** of the `cs2list<T>` class.

(b) Implement the `splice` member function. It should insert all nodes from `other` after the `k`-th entry of the `cs2list<T>` and before the `k+1`-st. If the list is shorter than `k` then all values from `other` should be appended to the end of the list. It must do this *without* creating any new nodes. At the end of the function call the list `other` must be empty.

15. 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 )
```

16. Our word counting program, discussed in class, created a map of the form

```
map< string, int > wc;
```

This map is an association between a word and the number of times it occurs in an input file. When we iterate through `wc`, we access the map entries in alphabetical order. Suppose instead we wanted the entries ordered by the number of times they occur, with words occurring the fewest times first and words occuring the most times last.

(a) One way to do this is to create another map,

```
map< int, list<string> > word_order;
```

where in each entry of the map the int is the number of occurrences and the list<string> gives the words that occurred that many times. For example, if "hello", "never", and "once" are the only words that occurred exactly 5 times in the input file then there should be an entry in the resulting map that contains the integer 5 and a list containing "hello", "never" and "once".

Write a function to create word_order from wc. Here is the prototype:

```
void alpha_to_occurrence( const map< string, int >& wc,
                          map< int, list<string> >& word_order)
```

(b) Implement a second version of function alpha_to_occurrence that produces a vector of lists of strings instead a map. After the function, the entry in the vector at location i should be the list of strings that occurred i times. (Of course, the vector entry at location 0 will be an empty list.) Thus, for the above example, word_order[5] should be a list containing "hello", "never" and "once". Here is the prototype

```
void alpha_to_occurrence( const map< string, int >& wc,
                          vector< list<string> >& word_order )
```

17. You are given a map, assoc, that associates a string (the “key”) with a list of strings (the “value”). Its declaration is

```
std::map< string, list<string> > assoc;
```

After processing input, assoc might contain, for example:

key	**value**
"apple"	"banana", "cat", "tiger", "tree", "car", "forest"
"banana"	"car", "angel", "simple"
"car"	"apple", "angel", "forest"
"forest"	"rain", "tree", "monkey", "ape"
"monkey"	"tree", "banana"
"tree"	"apple", "forest"
"zebra"	"simple", "apple", "car", "horse"

Observe that "forest" is in the list associated with "apple", but that "apple" is **not** in the list associated with "forest". Write a function that takes assoc as a parameter and outputs to std::cout all pairs of strings, s1 and s2, such that s1 is in the association list of s2 **and** s2 is in the association list of s1. For the example above, the output should be

```
apple tree
apple car
forest tree
```

You may assume that the list associated with any string does not contain repeated strings, and you may assume all strings are lower case.

Summary of the Primary Functionality of Maps

- A particular instance of a `map` is defined as

  ```
  map< key_type, value_type >  var_name
  ```

- Maps store *pairs* of "associated" values.

  ```
  pair< const key_type, value_type >
  ```

- Map iterators refer to pairs.
- Map `operator[]` is a function call taking a `key_type` as an argument and returning a reference to the associated `value_type` in the map.
- The `find` member function of maps is

  ```
  m.find( key );
  ```

  where `m` is the map object and `key` is the search key (of type `key_type`). It returns a map iterator, which is the `end` iterator if no pair in the map has `key` as its `first` value.

- The map `insert` member function

  ```
  m.insert( make_pair( key, value ) );
  ```

  returns a pair

  ```
  pair< map<key_type, value_type>::iterator, bool >
  ```

  where `second` value is `true` if the pair was inserted and `false` if a pair having the key was already in the map.

- Maps provide three different versions of the erase member function:
  - `void erase( iterator p )` — erase the pair referred to by iterator `p`.
  - `void erase( iterator first, iterator last )` — erase all pairs from the map starting at `first` and going up to, but not including, `last`.
  - `size_type erase( const key_type& k )` — erase the pair containing key `k`, returning either 0 or 1, depending on whether or not the key was in a pair in the map