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

Overview

- Test 2 will be held **Friday, March 23, 2007 2:00-3:30pm, West Hall Auditorium**. 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.
- Closed-book and closed-notes. **BUT,** you may bring a one-page (8.5x11) double-sided crib sheet without anything written or printed on it that you want. One potential use of this crib sheet is to outline the member functions of the list, vector and map container classes.
- Below are **many** relevant sample questions from previous tests. Solutions will be posted on-line.
- *How to study?*
  - Review lecture notes
  - Review and re-do lecture exercises, lab and homework problems.
  - Do as many of the practice problems below as you can. Focus on the ones that cause you trouble. Practice writing solutions using pencil (or pen) and paper.

Practice Problems

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;       // 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.

   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(); }
   ```

   This may also be done 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 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.

   Solution:

```
bool compare_enroll( const Course& c1, const Course& c2 )
d{
  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 `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.

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 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"
```

Solution: Version 1

```
void remove_non_alpha1( string& s )
{
  string result;
  for ( unsigned int i=0; i<s.size(); ++i )
    {
      if ( isalpha(s[i]) ) result.push_back(s[i]);
    }
  s = result;
}
```

Version 2:

```
void remove_non_alpha2( string& s )
{
  unsigned int j=0;
  for ( unsigned int i=0; i<s.size(); ++i )
    {
      if ( isalpha(s[i]) )
        {
          s[j] = s[i];
          ++j;
        }
    }
  s.resize(j);
}
```

7. 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;
}
```

You could also use `std::find`:

```
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 )
    {
      if ( std::find( p->second.begin(), p->second.end(), p->first )
             != p->second.end() )
        num ++ ;
    }
  return num;
}
```

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

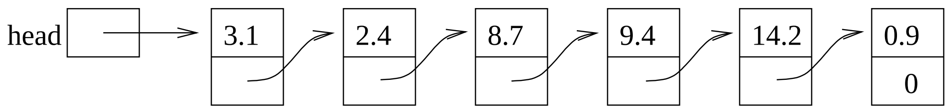
```
template <class T>
Node<T>* Sublist( Node<T>* head, int low, int high )
```

The `Node` class is:

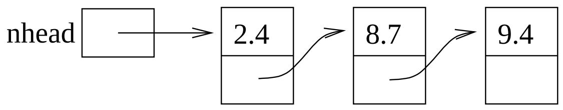
```
template <class T>
class Node {
public:
   T value;
   Node* next;
};
```

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



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.

Solution:

```
Node<T>* Sublist( Node<T>* head, int low, int high )
{
  //  Skip over the first low-1 nodes in the existing list,
  //  leaving p to point to the low-th node

  Node<T>*p = head;
  int i;
  for ( i=1; i<low; ++i ) p = p-> next;

  //  Make the new head node and make a pointer to the last node
  //  in the new list

  Node<T>* new_head = new Node<T>;
  new_head -> value = p -> value;
  Node<T> * last = new_head;

  //  Copy the remaining nodes, one at a time.

  for ( ++i, p = p->next; i<=high && p; ++i, p = p->next  ) {
    last -> next = new Node<T>;
    last -> next -> value = p -> value;
    last = last -> next;
  }
  last -> next = 0;
}
```

10. 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)
```

Solution: This exploits `operator[]`. Students could use a combination of insert and find as well.

```
void alpha_to_occurrence( const map< string, int >& wc,
                          map< int, list<string> >& word_order)
{
  word_order.clear();
  for ( map< string, int > :: const_iterator wc_p = wc.begin();
        wc_p != wc.end(); ++wc )
    word_order[ wc_p -> second ] . push_back( wc_p -> first );
}
```

(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 )
```

Solution:

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

  //  Find the maximum number of appearances and resize the vector
  unsigned int max_appearances = 0;
  for ( map<string,int>::const_iterator p=wc.begin(); p!=wc.end(); ++p )
    if ( p->second > max_appearances ) max_appearances = p->second;
  word_order.resize( max_appearances+1 );

  //  Now just add to the lists, as before.
  for ( map<string,int>::const_iterator p=wc.begin(); p!=wc.end(); ++p )
    word_order[p->second].push_back( p->first );
}
```

(c) 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
```

Solution:

```
bool common_char(const string &a, const string &b) {
  for (int i = 0; i < a.size(); i++)
    for (int j = 0; j < b.size(); j++)
      if (a[i] == b[j])
        return true;
  return false;
}

void filter(list<string>& words) {
  list<string>::iterator i,j;
  i = words.begin();
  while (i != words.end()) {
    j = i;
    j++;
    while (j != words.end()) {
      if (common_char(*i,*j)) {
        cout << *i << " removes " << *j << endl;
        j = words.erase(j);
      } else {
        j++;
      }
    }
    i++;
  }
}
```

11. You have been asked to help with a valet parking system for a big city hotel. The hotel must keep track of all of the cars currently stored in their parking garage and the names of the owners of each car. *Please read through the entire question before working on any of the subproblems.* Here is the simple `Car` class they have created to store the basic information about a car:

```
class Car {
public:
  // CONSTRUCTOR
  Car(const string &m, const string &c) : maker(m), color(c) {}
  // ACCESSORS
  const string& getMaker() const { return maker; }
  const string& getColor() const { return color; }
private:
  // REPRESENTATION
  string maker;
  string color;
};
```

The hotel staff have decided to build their parking valet system using a map between the cars and the owners. This map data structure will allow quick lookup of the owners for all the cars of a particular color and maker (e.g., the owners of all of the silver Hondas in the garage). For example, here is their data structure and how it is initialized to store data about the six cars currently in the garage.

```
map<Car,vector<string> > cars;
cars[Car("Honda","blue")].push_back("Cathy");
cars[Car("Honda","silver")].push_back("Fred");
cars[Car("Audi","silver")].push_back("Dan");
cars[Car("Toyota","green")].push_back("Alice");
cars[Car("Audi","silver")].push_back("Erin");
cars[Car("Honda","silver")].push_back("Bob");
```

The managers also need a function to create a report listing all of the cars in the garage. The statement:

```
print_cars(cars);
```

will result in this report being printed to the screen (`std::cout`):

```
People who drive a silver Audi:
  Dan
  Erin
People who drive a blue Honda:
  Cathy
People who drive a silver Honda:
  Fred
  Bob
People who drive a green Toyota:
  Alice
```

Note how the report is sorted alphabetically by maker, then by car color, and that the owners with similar cars are listed chronologically (the order in which they parked in the garage).

(a) In order for the `Car` class to be used as the first part of a map data structure, what additional non-member function is necessary? Write that function. Carefully specify the function prototype (using const & reference as appropriate). Use the example above as a guide.

Solution: We must define `operator<` for `Car` objects so that we can sort the keys of the map.

```
bool operator<(const Car &a, const Car &b) {
  return (a.getMaker() < b.getMaker() ||
          (a.getMaker() == b.getMaker() && a.getColor() < b.getColor()));
}
```

(b) Write the `print_cars` function. Part of your job is to correctly specify the prototype for this function. Be sure to use const and pass by reference as appropriate.

Solution:

```
void print_cars(const map<Car,vector<string> > &cars)
{
  map<Car,vector<string> >::const_iterator itr = cars.begin();
  while (itr != cars.end()) {
    Car c = itr->first;
    cout << "People who drive a " << c.getColor() << " "
         << c.getMaker() << ":" << endl;
    vector<string>::const_iterator itr2 = itr->second.begin();
    while (itr2 != itr->second.end()) {
      cout << "  " << *itr2 << endl;
      itr2++;
    }
    itr++;
  }
}
```

(c) When guests pick up their cars from the garage, the data structure must be correctly updated to reflect this change. The `remove_car` function returns true if the specified car is present in the garage and false otherwise.

```
  bool success;
  success = remove_car(cars, "Erin", "silver", "Audi");
  assert (success == true);
  success = remove_car(cars, "Cathy", "blue", "Honda");
  assert (success == true);
  success = remove_car(cars, "Sally", "green", "Toyota");
  assert (success == false);
```

After executing the above statements the `cars` data structure will print out like this:

```
  People who drive a silver Audi:
    Dan
  People who drive a silver Honda:
    Fred
    Bob
  People who drive a green Toyota:
    Alice
```

Note that once the only blue Honda stored in the garage has been removed, this color/maker combination is completely removed from the data structure.

Specify the prototype and implement the `remove_car` function.

Solution:

```
bool remove_car(map<Car,vector<string> > &cars,
                const string &name,
                const string &color,
                const string &maker)
{
  map<Car,vector<string> >::iterator itr = cars.find(Car(maker,color));
  if (itr == cars.end()) return false;
  if (itr->second.size() == 1 && itr->second[0] == name) {
    cars.erase(Car(maker,color));
    return true;
  }
  for (int i = 0; i < itr->second.size(); i++) {
    if (itr->second[i] == name) {
      itr->second.erase(itr->second.begin() + i);
      return true;
    }
  }
  return false;
}
```