Computer Science II — CSci 1200
Test 1 Overview and Practice

Overview

- Test 1 will be held **Tuesday, February 13, 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.
- Purpose: Check your understanding of the basics of (a) C++, (b) solving small computational problems, (c) the standard library (streams, strings and vectors), and (d) memory management.
- Coverage: Lectures 1-7, Labs 1-4, HW 1-3.
- 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.
- 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 vector of char in reverse order.

   Solution:

   ```
   //  assume the string is s and s has been initialized
   vector<char> result;
   for ( int i=s.size()-1; i>=0; --i )
       result.push_back( s[i] );
   ```

2. 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;
}
```

3. 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 );
}
```

4. Write a function called `less_string` that mimics the effect of the `<` operator on strings. In other words, given strings `a` and `b`, `less_string( a, b)` should return `true` if and only if `a < b`. Of course, you may use `<` on individual characters in the string. Start by getting the function prototype correct.

   Here are examples of pairs for which your function should return true:

```
a = "abc",  b = "abd"
a = "cab",  b = "cabbage"
a = "christine",  b = "christopher"
```

Solution:

```
bool
less_string( const string& a, constr string& b )
{
  unsigned int min_length = a.size();
  if ( b.size() < min_length ) min_length = b.size();
  while ( i < min_length )
    {
      if ( a[i] < b[i] )
        return true;
      else if (a[i] > b[i] )
        return false;
      ++ i;
    }
  return a.size() < b.size();
}
```

5. What is the output from the following program? We strongly suggest that you draw the contents of the vectors to help you visualize what is happening.

```
void more_confused( vector<int> a,  vector<int> & b )
{
  for ( unsigned int i=0; i<2; ++i )
    {
      int temp = a[i];
      a[i] = b[i];
      b[i] = temp;
    }
  cout << "1: ";
  for ( unsigned int i=0; i<a.size(); ++i )
    cout << a[i] << " ";
  cout << endl;
  cout << "2: ";
  for ( unsigned int i=0; i<b.size(); ++i )
    cout << b[i] << " ";
  cout << endl;
}

int
main()
{
  vector<int> a, b;
  a.push_back(1); a.push_back(3); a.push_back(5);
  b.push_back(2); b.push_back(4);

  more_confused( a, b );
```

```
  a[0] = 7; a[1] = 9;
  more_confused( b, a );

  cout << "3: ";
  for ( unsigned int i=0; i<a.size(); ++i )
    cout << a[i] << " ";
  cout << endl;

  cout << "4: ";
  for ( unsigned int i=0; i<b.size(); ++i )
    cout << b[i] << " ";
  cout << endl;

  return 0;
}
```

Solution

```
1: 2 4 5
2: 1 3
1: 7 9
2: 1 3 5
3: 1 3 5
4: 1 3
```

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

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

   then the resulting vector of negative numbers should contain

   `-1.3, -4.5, -9.1`

   and the resulting vector 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 vector<double>& original,
                   vector<double>& negatives,    // passing by reference is required
                   vector<double>& positives )   // passing by reference is required
{
  negatives.clear();  positives.clear();   // not necessary
  for ( unsigned int i=0; i<original.size(); ++i )
    {
       if ( original[i] < 0 )
           negatives.push_back( original[i] );
       else if ( original[i] > 0 )
           positives.push_back( original[i] );
    }
}
```

7. Give an order notation ("O") estimate of the worst-case number of operations required by the following sorting function. Briefly justify your answer:

```
void
ASort( vector<double>& A )
{
  int n = A.size();
  for( int i=0; i<n-1; ++i )
    {
      //  Find index of next smallest value
      int small_index = i;
      for ( int j=i+1; j<n; ++j )
        {
          if ( A[j] < A[small_index] )
            small_index = j;
        }

      //  Swap with A[i]
      double temp = A[i];
      A[i] = A[small_index];
      A[small_index] = temp;
    }
}
```

Solution: This is $O(n^2)$. Clearly the outer loop requires $n-1$ iterations. Each iteration of the outer loop has a constant number of operation before and after the inner loop. The inner loop has $n-i-1$ iterations, which is at most $n-1$. Each iteration requires constant time. Thus, each iteration of the outer loop is $O(n)$, giving a total of $O(n^2)$ for all iterations.

8. Write a function that takes an array of ints and copies the **even integers** into a dynamically-allocated array of ints, storing the values in **in reverse order**. For example, given an array containing the 12 values

```
5, -1, 4, 0, 13, 27, 98, 17, 97, 24, 98, 89
```

the resulting array should contain the values (in order)

```
98, 24, 98, 0, 4
```

after the function is completed. When doing dynamic allocation, you may only allocate enough space to store the actual numbers, and no more.

The return type of the function must be `void`. Start by specifying the function prototype. Think carefully about the parameters needed and their types. Repeat the problem using (a) pointers instead of array subscripting and (b) returning the even integers in a vector instead of an array. Give an "O" estimate for each

Solution: One challenge here is how to pass the resulting array. Two reference parameters are needed. One is a pointer to the dynamically-allocated array and the second is an integer giving the size of the array. Here is the result:

```
void
even_integers_reversed( int a[], int n, int* & b, int & m )
{
  m = 0;
  for ( unsigned int i=0; i<n; ++i )  // count even values
    if ( a[i] % 2 == 0 ) ++m;
  b = new int[m];
  int i=0;                            //  index into array pointed to by b
  for ( int j=n-1; j>=0; j-- )        //  go backwards through a
    if ( a[j] % 2 == 0 )
      {
         b[i] = a[j];
         ++ i;
      }
}
```

Here is the pointer version

```
void
even_integers_reversed( int a[], int n, int* & b, int & m )
{
  m = 0;
  for ( int* p = a; p < a+n; ++p )   // count even values
    if ( *p % 2 == 0 ) ++m;
  b = new int[m];
  int * q = b;                           //  index into array pointed to by b
  for ( int* p = a+n-1; p>=a; p-- )      //  go backwards through a
    if ( *p % 2 == 0 )
      {
         *q = *p;
         ++ q;
      }
}
```

The vector version requires that we go backward through a, but only one time.

```
void
even_integers_reversed( int a[], int n, vector<int> & b )
{
  b.clear();
  for ( int j=n-1; j>=0; j-- )        //  go backwards through a
    if ( a[j] % 2 == 0 ) b.push_back( a[j] );
}
```

All versions are $O(n)$. In the first two cases there are two separate (non-nested) loops through array a. In each loop, a constant amount of work is done and there are n loop iterations. This gives $O(n)$ per loop, but since they are in sequence, the total cost is $O(n)$ overall. The argument is even simpler for the vector version as long as we can assume `push_back` is constant time, which it is on average.

9. Consider the following declaration of a `Point` class, including an associated non-member function:

```
class Point {
 public:
  Point();
  Point( double in_x, double in_y, double in_z );
  void get( double & x, double & y, double & z ) const;
  void set( double x, double y, double z );
  bool dominates( const Point & other );
 private:
  double px, py, pz;
};

bool dominates_v2( const Point & left, const Point & right );
```

   (a) Provide the implementation of the default constructor — the constructor that takes no arguments. It should assign 0 to each of the member variables.

   Solution:

```
Point::Point()
{
  xp = yp = zp = 0.0;
}
```

   (b) The member function `dominates` should return `true` whenever each of the point's coordinates is greater than or equal to each of the corresponding coordinates in the other point. For example, given

```
Point p( 1.5, 5.0, -1 );
Point q( 1.4, 5.0, -3 );
Point r( -3, 8.1, -7 );
```

   Then

```
p.dominates( q )
```

should return `true` but the function calls

```
p.dominates( r )    r.dominates( q )    q.dominates(r)
```

should each return false. Write member function `dominates`.

Solution:

```
bool
Point::dominates( const Point& other )
{
  return xp >= other.xp && yp >= other.yp && zp >= other.zp;
}
```

(c) The function `dominates_v2` should be a non-member function version of `dominates`. Its behavior should be essentially the same as the member function version, so that for the `Point` objects defined in (b),

```
dominates_v2( p, q )
```

should return `true` but the function calls

```
dominates_v2( p, r )    dominates_v2( r, q )    dominates_v2( q, r)
```

should each return `true`. Write `dominates_v2`.

Solution:

```
bool dominates_v2( const Point & left, const Point & right )
{
  double xpl, ypl, zpl;
  left.get( xpl, ypl, zpl );
  double xpr, ypr, zpr;
  right.get( xpr, ypr, zpr );
  return xpl >= xpr && ypl >= ypr && zpl >= zpr;
}
```

10. Consider the following start to a class declaration:

```
class bar {
public:
  bar( int in_x, double in_y, const vector<string>& in_z )
    : x(in_x), y(in_y), z(in_z) {}

private:
  int x;
  double y;
  vector<string> z;
};
```

This class is incomplete because no member functions are defined.

(a) Write a member function of class `bar` called *OrderZ* that sorts the member variable `z` into increasing order. Show both its prototype in `bar` and its implementation, outside of the declaration for `bar`.

Solution: Below you will find a completed implementation of the `bar` class, including all needed member functions.

(b) Write a function that takes a vector of `bar` objects and re-arranges them so that the objects are ordered by decreasing value of `x`, and by increasing `y` for `bar`'s with equal values of `x`. Give two different solutions. One solution uses an `operator<` on `bar` objects and the other uses a non-operator comparison function on `bar` objects. In both cases, you will need to add member functions to `bar`. Be sure to include these in your solution.

Solution:

```
// should be in bar.h

class bar {
public:
  bar( int in_x, double in_y, const vector<string>& in_z )
    : x(in_x), y(in_y), z(in_z) {}

  void OrderZ();

  int getx() const { return x; }
  double gety() const { return y; }
private:
  int x;
  double y;
  vector<string> z;
};

bool operator< ( const bar& b1, const bar& b2 );
bool compare_bar( const bar& b1, const bar& b2 );

////////////////////////////////////////////

// should be in bar.cpp

void bar :: OrderZ()
{
  sort( z.begin(), z.end() );
}


bool operator< ( const bar& b1, const bar& b2 )
{
  return b1.getx() > b2.getx() ||
    ( b1.getx() == b2.getx() && b1.gety() < b2.gety() );
}
```

```
bool compare_bar( const bar& b1, const bar& b2 )
{
  return b1.getx() > b2.getx() ||
    ( b1.getx() == b2.getx() && b1.gety() < b2.gety() );
}

//////////////////////////////////////////////////////////

void
order_bars_1( vector<bar> & bars )
{
  sort( bars.begin(), bars.end() );
}


void
order_bars_2( vector<bar> & bars )
{
  sort( bars.begin(), bars.end(), compare_bar );
}
```

11. 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;
```

12. 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
```

13. 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
```

Solution: The confusing part of the problem is that the new vector being created is not the current object — the object the member function is called on. 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 = m_size;
  new_vec.m_data = new T[new_vec.m_size];
  for ( T * p = new_vec.m_data, *q = m_data+m_size-1;
        q >= m_data; p++, q-- )
    *p = *q;
  return new_vec;
}
```

14. Here is a program that is *supposed* to detect and output each distinct integer that occurs 2 or more times in an input sequence. Unfortunately, there is a small problem with this program causing it to work incorrectly for some inputs.

```
int main() {
  vector<int> input;      // stores all the numbers
  vector<int> duplicates; // stores all numbers that appear more than once

  // read in the input & search for duplicates
  int x;
  while (cin >> x)
    {
      input.push_back(x);
      for (int i = 0; i < input.size()-1; i++)
        {
          if (x == input[i])
          duplicates.push_back(x);
        }
    }

  // print out the duplicates
  if (duplicates.size() > 0)
    {
      cout << "These numbers appeared more than once in the input: " << endl;
      for (int i = 0; i < duplicates.size(); i++)
        cout << duplicates[i] << " ";
      cout << endl;
    }
}
```

(a) First let's look at a test case for which the program works correctly. What is printed on the screen when the user inputs this sequence:

```
1 2 1 3 4 3 5 2
```

Solution:

```
These numbers appeared more than once in the input:
1 3 2
```

(b) Now, give an example sequence of integers where the program behaves incorrectly. What is the behavior/output for your test case, and what should the output be?

Solution: In any sequence with an integer that appears 3 or more times, that integer will be listed multiple times. For example, the sentence:

```
1 2 1 3 4 3 5 2 1
```

will have this output:

```
These numbers appeared more than once in the input:
1 3 2 1 1
```

and the output should be:

```
These numbers appeared more than once in the input:
1 3 2
```

(c) Describe how to modify the code to fix the problem. Rewrite the part of the program with the bug so that it works correctly for all sequences of integers.

Solution: Before we add an element to the duplicate list, we need to verify that it is not already in the list.

```
if (x == input[i])
 {
    bool found = false;
    for (int j = 0; j < duplicates.size(); j++)
      {
        if (x == duplicates[j])
          found = true;
      }
    if (!found)
      duplicates.push_back(x);
  }
```

15. 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
```

16. In this problem you will implement a simple class named `Major` to store information about students and their declared majors. Since students often change their minds, your class will need to handle these changes and keep track of how many times a student changed majors. Please carefully read through all of the information below before working on the class design.

    Here are several examples of how we will create and initialize `Major` objects:

    ```
    Major sally("Sally");
    Major fred("Fred");
    Major bob("Bob");
    Major alice("Alice");
    ```

    Initially each student's major is listed as undeclared, but they can change their major with the `declareMajor` member function:

    ```
    sally.declareMajor("Economics");
    fred.declareMajor("Information Technology");
    bob.declareMajor("Chemistry");
    sally.declareMajor("Information Technology");
    bob.declareMajor("Psychology");
    bob.declareMajor("Biology");
    ```

    We also need various accessor functions to get the student's name, their major, and the number of times they changed their major. Here's an example use of these member functions:

    ```
    cout << bob.getName() << " changed majors " << bob.numChanges()
         << " time(s) and is currently majoring in " << bob.getMajor() << "." << endl;
    ```

    Which will result in this output to the screen:

    ```
    Bob changed majors 3 time(s) and is currently majoring in Biology.
    ```

    We can also store multiple `Major` objects in a `vector` to organize the student registration database:

    ```
    vector<Major> all_students;
    all_students.push_back(sally);
    all_students.push_back(fred);
    all_students.push_back(bob);
    all_students.push_back(alice);
    ```

    In the third part of this problem you will write code to sort and output this database with the students grouped by their current major and sorted alphabetically:

```
Biology
  Bob
Information Technology
  Fred
  Sally
Undeclared
  Alice
```

(a) Using the foregoing sample code as your guide, write the class declaration for the `Major` object. That is, write the *header file* (`major.h`) for this class. You don't need to worry about the `#include` lines or other pre-processor directives. Decide how you are going to store the information for each object. Focus on getting the member function prototypes correct. Use `const` and call by reference where appropriate. Make sure you label what parts of the class are `public` and `private`. Don't include any of the member function implementations here (even if they are just one line). Save the implementation for the next part.

Solution:

```
class Major {
public:
  // CONSTRUCTOR
  Major(const string& name);

  // ACCESSORS
  const string& getName() const;
  const string& getMajor() const;
  int numChanges() const;

  // MODIFIER
  void declareMajor(const string &major);

private:
  // REPRESENTATION
  string m_name;
  string m_major;
  int m_num_changes;
};
```

(b) Now implement the constructor and member functions you declared in the previous part, as they would appear in the corresponding `major.cpp` file.

Solution:

```
Major::Major(const string &name) {
  m_name = name;
  m_major = "Undeclared";
  m_num_changes = 0;
}
```

```
const string& Major::getName() const {
  return m_name;
}

const string& Major::getMajor() const {
  return m_major;
}

int Major::numChanges() const {
  return m_num_changes;
}

void Major::declareMajor(const string &major) {
  m_major = major;
  m_num_changes++;
}
```

(c) Now we need to prepare the student lists for each department. Given the `all_students` vector that stores all of the students, your task is to write code that sorts the students into groups by major and then alphabetically within each group. This can be done with a single call to the `sort` function for vectors. First write the helper function you'll need for the sort operation.

Solution:

```
bool major_then_alphabetical(const Major &m1, const Major &m2) {
  return (m1.getMajor() < m2.getMajor()) ||
         ((m1.getMajor() == m2.getMajor()) && (m1.getName() < m2.getName()));
}
```

(d) Finally, write the code that uses this helper function to sort and then output the registration database in this format:

```
Biology
  Bob
Information Technology
  Fred
  Sally
Undeclared
  Alice
```

Solution:

```
sort(all_students.begin(), all_students.end(), major_then_alphabetical);
string current_major = "";
for (int i = 0; i < all_students.size(); i++) {
  if (all_students[i].getMajor() != current_major) {
    // detect when we switch to the next major
    current_major = all_students[i].getMajor();
    cout << current_major << endl;
```

```
  }
  cout << "  " << all_students[i].getName() << endl;
}
```