CSE 303 Final Exam

March 16, 2009

Name _____

The exam is closed book, except that you may have a single page of hand-written notes for reference, plus the single page of notes from the midterm.

If you have questions during the exam, raise your hand and someone will come to help. Stay seated.

Please wait to turn the page until everyone has their exam and you have been told to begin.

1	/ 12
2	/ 12
3	/ 12
4	/ 12
5	/ 12
6	/ 12
7	/ 16
8	/ 12
Total	/ 100

Question 1. (12 points) The assert macro in C and the assert statement in Java can be used during debugging to check for errors during execution. Some of the things that we'd like to check can be done very efficiently (i.e., check whether a variable is 0), some can be done but take significantly more time (check that all of the elements in a large array are 0), and some cannot be done at all because they are not feasible.

For each of the following possibilities, indicate whether we can check it in an assertion and, if so, whether it is something that is simple (fast) to check or not simple (potentially slow). Put an X in the correct column to the left of each choice.

Several of the questions refer to things that might have been useful to check during the memory manager (getmem/freemem) project.

Fast?	Slow?	Can't?	What
			In a call freemem(p), p is not null
			In a call freemem(p), p has not been freed previously (is not on the free list)
			In a call freemem(p), the pointer value p refers to a block of storage previously allocated by getmem
			In a call freemem(p), the freed block is not used again by the caller
			The free list is properly sorted by ascending block address
			After splitting a free block into two blocks, one that is returned to the caller and the other to be left on the free list, the block left on the list has a size greater than 8

Question 2. (12 points) What output does the following C program produce?

```
#include <stdio.h>

#define TWO 2

#define VII 3+4

#define TIMES(x,y) x*y

#define PROD(x,y) (x*y)

#define MULT(x,y) (x)*(y)

int main() {
  int a,b,c;
  a = PROD(VII,VII);
  b = MULT(VII,TWO+TWO);
  c = TIMES(3,VII);
  printf("%d %d %d\n", a, b, c);
  return 0;
}
```

Question 3. (12 points) Two questions about version control, in particular svn.

(a) You are giving a demo of a project that is stored in a svn repository. The code works fine on the test machine in the lab, but it crashes with an old bug when you try to run it on the demo laptop. What is most likely to be the problem, and how do you fix it, preferably quickly?

(b) Subversion (svn) uses an optimistic model for handling updates by multiple users. Instead of obtaining exclusive access to a file before updating it, any user can modify any file in the project without having to obtain permission first. But that potentially allows different users to introduce changes into a single file that conflict with each other. What happens when conflicting changes are made to a file? How and when is the problem detected, and what needs to be done to resolve the conflict? (Be brief – a couple of sentences should be enough to get the idea across.)

Question 4. (12 points) The statistics produced by the `bench` program for the memory manager project provide not only performance information, but also can give clues to how well the various functions like `getmem` and `freemem` are working.

Suppose we run the bench program and ask it to use a 50% probability to distribute calls between `getmem` and `freemem`, i.e., both `getmem` and `freemem` will be called roughly the same number of times as the program runs. Here is the output produced for a sample run, showing only the number of blocks on the free list and their average size as the run progresses.

```
10%  done: # blocks on freelist: 516 Avg. size of blocks: 191
20%  done: # blocks on freelist: 1028 Avg. size of blocks: 141
30%  done: # blocks on freelist: 1530 Avg. size of blocks: 101
40%  done: # blocks on freelist: 2020 Avg. size of blocks: 89
50%  done: # blocks on freelist: 2522 Avg. size of blocks: 61
60%  done: # blocks on freelist: 3031 Avg. size of blocks: 63
70%  done: # blocks on freelist: 3532 Avg. size of blocks: 50
80%  done: # blocks on freelist: 4028 Avg. size of blocks: 47
90%  done: # blocks on freelist: 4506 Avg. size of blocks: 40
100% done: # blocks on freelist: 5017 Avg. size of blocks: 37
```

Other runs of bench using the same 50% probability and the same getmem/freemem implementation yield similar numbers even when the random number seed is different.

Looking at this information, what can you conclude about this particular implementation of `getmem` and `freemem`? Do the numbers look reasonable? If not, what might be wrong? What in the data leads you to your conclusion? (Please try to be brief.)

Question 5. (12 points) To demonstrate C++ classes we had an example that described real estate objects in terms of a basic `Property` class, a class `Land` that extended it, and a main program that created objects and called their methods. For this question we'd like to set up a `Makefile` to build this program, recompiling things only when needed.

For reference, here is an outline of the various header and implementation files showing the relationships between them.

```
**********************
* Property.h         *
**********************
#ifndef PROPERTY_H
#define PROPERTY_H

class Property {
  ...
};

#endif
```

```
**********************
* Land.h             *
**********************
#ifndef LAND_H
#define LAND_H

#include "Property.h"

class Land : public Property {
  ...
};

#endif
```

```
**********************
* Property.cc        *
**********************

#include "Property.h"

// definitions of Property
// member functions
```

```
**********************
* Land.cc            *
**********************

#include "Land.h"

// Definitions of Land
// member functions
```

```
**********************
* Main.cc            *
**********************

#include "Property.h"
#include "Land.h"

int main() {
  ...
}
```

(You may remove this page for reference if you wish.)

Question 5. (cont.) One of the summer interns came up with this `Makefile` for the project which, unfortunately, doesn't work well. Find the problems in the `Makefile`, briefly describe what's wrong, and indicate how to fix things by either making corrections in place (cross out unneeded things and write in corrections), or adding necessary corrections at the end. The corrected `Makefile` should create the program `test` when make is run without arguments, and should only recompile the minimum number of files needed to do so.

```
Land.o:  Land.cc  Property.cc  Land.h

      gcc -Wall -g -c Land.cc

Property.o:  Property.cc  Property.h

      gcc -Wall -g -c Property.cc

Main.o:  Main.cc  Property.cc  Land.cc

      gcc -Wall -g -c Main.cc

test:  Main.o  Property.o  Land.o

      gcc –Wall –g –o test Main.o Property.o Land.o
```

Question 6. (12 points) The C++ "what does this print?" question. Suppose we have the following C++ classes and main program:

```
#include <iostream>
using namespace std;

class Bovine {
public:
  void speak() { cout << "harumph" << endl; eat(); }
  void eat()   { cout << "gulp" << endl; }
};

class Cow: public Bovine {
public:
  virtual void eat()   { cout << "chew" << endl; }
  virtual void speak() { cout << "moo" << endl; eat(); }
};


int main() {
  Bovine * fred = new Cow();
  fred->speak();
  fred->eat();

  Cow * clarabell = new Cow();
  clarabell->speak();
  clarabell->eat();

  delete fred;
  delete clarabell;
  return 0;
}
```

(In the above code, the function implementations are included as part of the classes, instead of being written as separate functions elsewhere. This is legal C++ and compiles and executes without errors.) What output is produced when this program is executed?

Question 7. (16 points) Although people used slightly different trie data structures for the text translation project, assume for this problem that a trie node is represented as follows.

```
struct tnode {
  char * word;       // null-terminated word associated with this node
                     //    or NULL if no string attached to this node
  struct tnode *child[10]; // pointers to subtrees of this node.  If a
}                          // subtree is empty, its pointer is NULL
```

You may assume that `child[2]` through `child[9]` are the child pointers for the digits 2-9, and `child[0]` is the '#' link, although this information may not actually be needed for this particular problem.

Implement function `tclone` below so it allocates and makes a complete copy of the tree whose root node is given as its argument, and returns a pointer to the root of the new tree. The clone should contain the same number of `tnode`s as the original tree, linked the same way, and should contain a duplicate copy of each string in the original tree, linked to the corresponding `tnode` in the copy.

You may define additional functions if it helps, but you should think first before you code. The sample solution is not particularly long or complex. You should assume that any relevant header files for things like `NULL` or the string library are already included, and you do not need to write the corresponding `#include`s.

(write your solution on the next page)

Question 7. (cont.) Reminder: a trie node is represented as follows:

```
struct tnode {
  char * word;       // null-terminated word associated with this node
                     //    or NULL if no string attached to this node
  struct tnode *child[10]; // pointers to subtrees of this node.  If a
}                          // subtree is empty, the pointer is NULL
```

Implement your function below:

```
/* return a complete, new copy of the trie with root r */
struct tnode * tclone(struct tnode * r) {









}
```

Question 8. (12 points) Your company is building a new online game. Each time a player finishes playing the game, we want to check the player's score and see if it is higher than any previous score. If it is, we want to remember the new highest score and the player's id number. In addition, there is a function that will print the highest score and the high-scoring player's id number. The C code looks like this.

```
int max_score;     // highest score so far
int max_id;        // id of player achieving highest score

/* If this is a new high score, record the score and the player's id */
void update(int score, int id) {
  if (score > max_score) {
    max_score = score;
    max_id = id;
  }
}

/* Print highest score and the high-scoring player's id */
void print() {
  printf("max score is %d\n", max_score);
  printf("scored by player %d\n", max_id);
}
```

The question is whether this code will work properly if it is used in a multi-threaded system where more than one thread can be executing these functions concurrently. For each of the following three scenarios involving two threads, explain whether the code will always produce the correct results, or if not, what could go wrong. Briefly justify each of your answers.

You should assume that each individual line of code (assignment statements, conditional test, `printf`) will always execute without interruption by another thread. However, control can be transferred between threads at any time between any two lines of code.

(Continued on next page. Feel free to detach this page if it is helpful.)

Question 8 (cont.)

(a) Two threads both execute function `update` to record scores for two different players.

(b) One thread executes function `update` to record a player's score. A second thread executes function `print` to display the highest score and player id.

(c) Two threads both execute function `print` to display the highest score and player id.