Name:____

CSE 303, Spring 2005, Final Examination
7 June 2005

Please do not turn the page until everyone is ready.

Rules:

- The exam is closed-book, closed-note, except for one side of one 8.5x11in piece of paper.
- **Please stop promptly at 4:20.**
- You can rip apart the pages, but please write your name on each page.
- There are **90** total points, distributed **unevenly** among 8 questions (which all have multiple parts).
- When writing code, style matters, but don't worry about indentation.

Advice:

- Read questions carefully. Understand a question before you start writing.
- Write down thoughts and intermediate steps so you can get partial credit.
- The questions are not necessarily in order of difficulty. **Skip around.**
- If you have questions, ask.
- Relax. You are here to learn.

Name:____

1. Consider this C program, which compiles without warning, but crashes when run:

```
int factorial(int x) {
  if(x==1)
    return 1;
  return x * factorial(x-1);
}
int main(int argc, char**argv) {
  factorial(0);
}
```

   (a) **3pts** Looking at the source code, why does the program crash?

   (b) **6pts** What would happen if you used `gdb` to run this program? Without looking at the source code, what `gdb` commands would you use? What would you be able to conclude?

   Solution:

   (a) It overflows the stack: `factorial` will call itself recursively billions of times (assume 32-bit ints) and there is not enough room for a stack that large. When the stack reaches inaccessible memory, a segmentation-fault occurs. `factorial` does not expect arguments less than 1.

   (b) `gdb` would detect the segmentation fault and allow inspection of the program's state. Using the `backtrace` command would show that the stack is full of thousands of recursive calls to `factorial`, indicating that the problem almost certainly a stack overflow resulting from the way `factorial` is written and/or called.

Name:____

2. Suppose a C program includes this code, which includes a loop that is *useless*. Assume that `x` and `y` are valid pointers to legal strings (that end in `'\0'`) .

```
int f(char *x, char* y) {
  int i=0;
  for(; i < 10000000; ++i)
    strcmp(x,y);
  return 7;
}
```

In the 3 separate problems below, suppose you use `gprof` to profile this program. *You must give a* ***different*** *answer for each problem.*

(a) **5pts** The time samples from `gprof` show that the program spends most of its time in `strcmp`, but removing the loop from `f` has no noticeable effect on performance. What is the most likely explanation?

(b) **5pts** The call counts from `gprof` show that `strcmp` is called much more than any other function and 60% of the calls to `strcmp` come from `f`, but removing the loop from `f` has no noticeable effect on performance. What is the most likely explanation?

(c) **3pts** The time samples from `gprof` show that the program spends most of its time in `strcmp` *and* the call counts from `gprof` show that `strcmp` is called much more than any other function and 60% of the calls to `strcmp` come from `f`, but removing the loop from `f` *still* has no noticeable effect on performance. What is the most likely explanation?

Solution:

(a) Most of the calls to `strcmp` are from other sources. Perhaps `f` is never even called when the program runs.

(b) Although `strcmp` is called a lot, it is not where the program spends most of its time. Perhaps a function that is called relatively few times takes a long time to execute because it has long loops.

(c) Some calls to `strcmp` take longer than others and the calls from `f` are relatively quick. For example, suppose the arguments to `f` are short (e.g., one-character long), but the other calls to `strcmp` pass very long strings.

Name:____

3. Consider this type definition for *trees of integers* in C and 3 functions that allegedly deallocate the space for a tree:

```
#include <stdlib.h>
struct Tree {
  int val;
  struct Tree * left;
  struct Tree * right;
};
void free_tree_1(struct Tree * t) {
  if(t == NULL)
    return;
  free(t);
}
void free_tree_2(struct Tree * t) {
  if(t == NULL)
    return;
  free(t);
  free_tree_2(t->left);
  free_tree_2(t->right);
}
void free_tree_3(struct Tree * t) {
  if(t == NULL)
    return;
  free_tree_3(t->left);
  free_tree_3(t->right);
  free(t);
}
```

(a) **8pts** Explain which of the three functions is the best. Explain why the other two are not well-written.

(b) **4pts** Explain what assumption(s) the best function is implicitly making and how the function is wrong if the assumption(s) are violated.

Solution:

(a) The third function is best. The first creates space leaks if the tree's children are not otherwise reachable. The second has dangling-pointer dereferences; technically you may not use `t->left` or `t->right` after the object `t` points to is deallocated. The third function correctly frees the subtrees and then frees the root node.

(b) In addition to assuming all the `struct Tree *` pointers point to live heap-allocated objects of type `struct Tree`, the third function assumes the pointers actually form a tree. Put another way, it assumes that there is no sharing; all the pointers are unique. If two pointers in the alleged tree point to the same `struct Tree`, then the function will attempt to deallocate the object twice, which is an error.

Name:____

4. Here are the contents of three files that together form a program:

   - a.c:

     ```
     void f(int* x, int* y) { *y = *x; }
     ```

   - a.h:

     ```
     #ifndef A_H
     #define A_H
     void f(int*);
     #endif
     ```

   - b.c:

     ```
     #include <a.h>
     int main(int argc, char**argv) {
       int x;
       f(&x);
       return 0;
     }
     ```

   (a) **2pts** Why is this program incorrect?

   (b) **4pts** Will `gcc -c a.c; gcc -c b.c; gcc a.o b.o` create an executable `a.out` or will there be compiler errors? Explain.

   (c) **4pts** To catch this program's error, would it help to have `a.c` include `a.h`? Explain.

   (d) **4pts** To catch this program's error, would it help to use a Makefile that recompiles `a.c` and `b.c` whenever `a.h` changes? Explain.

   Solution:

   (a) Because `f` expect two arguments, but `main` passes it only one.

   (b) It will create an executable. Each file is compiled separately and typechecks, but they make different assumptions about how many arguments `f` takes. The linker will not catch this error for C code.

   (c) Yes, now compiling `a.c` will fail (or at least give a warning) because the definition for `f` does not match its earlier declaration.

   (d) No, recompiling `a.c` if `a.h` changes does not detect the error; compiling `a.c` will still succeed.

Name:____

5. Here are the contents of 4 files:

   - a.java: `class A { static boolean f() { return true; } }`
   - b.java:

     ```
     class B { public static void main(String[] args) {
                 if(args.length < 3)
                   A.f();
             } }
     ```

   - a.c: `int f() { return 1; }`
   - b.c:

     ```
     int f(); // declaration of function defined in another file
     int main(int argc, char **argv) {
       if(argc < 3)
          f();
       return 0;
     }
     ```

   For each of the following command sequences, explain whether the last command would succeed or cause some sort of error. **3pts each**

   (a)
   ```
   javac a.java
   javac b.java
   rm A.class
   java B 1 2 3 4
   ```
   (b)
   ```
   javac a.java
   javac b.java
   rm A.class
   java B 1
   ```
   (c)
   ```
   gcc -c a.c
   gcc -c b.c
   gcc -o prog a.o b.o
   rm a.o
   ./prog 1 2 3 4
   ```
   (d)
   ```
   gcc -c a.c
   gcc -c b.c
   gcc -o prog a.o b.o
   rm a.o
   ./prog 1
   ```

   Solution:

   (a) Succeed: Nothing in `A` is actually needed at run-time, so the class-loader never looks for `A.class`.

   (b) "Class not found" error: When the call to `A.f` is reached, the class-loader will look for `A.class`, not find it, and raise an exception.

   (c) Succeed: `gcc` uses `.o` files and the linker to make an executable. After that point, the `.o` files are unnecessary; all the code is in the executable.

   (d) Succeed: Same reason as previous question.

Name:____

6. Consider this Java code, assuming that `assert` evaluates its argument and raises an exception if the result is false (i.e., "the assertion fails"). (Assume there is only one thread and assertions are "enabled".)

```
class List {
  Object head;
  List   tail;
  List(Object h, List t) { head = h; tail = t; }
}
final class BackupList { // final means no subclasses, so that is not an issue
  private List lst = null;
  private List backup = null;
  public List get() { return lst; }
  public void add(Object obj) {
    assert(lst.tail == backup); // (1)
    backup = lst;
    lst = new List(obj,lst);
    assert(lst.tail == backup); // (2)
  }
}
```

(a) **3pts** A bad thing will happen when you call the `add` method on a `BackupList`. What is the bad thing and how would you change the line marked `(1)` to avoid it? (Your result should still check what `(1)` is attempting to check.)

(b) **3pts** Would you make an analogous change to line `(2)`. Why or why not?

(c) **3pts** Given your change to `(1)`, can the assertion at line `(1)` fail? If so, how? If not, why not?

(d) **3pts** Can the assertion at line `(2)` fail? If so, how? If not, why not?

Solution:

(a) The call to `add` will throw a NullPointerException on line (1) because `lst` starts `null`. A better assertion is `assert(lst==null || lst.tail==backup)`.

(b) No, when control reaches (2), `lst` cannot be `null`.

(c) Yes, it can fail because `get` returns a reference to the list in `lst`. So a client could set `lst.tail` to a different list than the one it held after a call to `add`.

(d) No, it cannot fail. The previous two assignment statements ensure `lst.tail` and `backup` both hold the list that was held in `lst` when `add` was called.

Name:____

7. This problem asks you to design a Makefile and version-control scheme for automatically generating documentation for Java code.

   Scenario:

   - Assume `a.java` defines one class `A`, and `b.java` defines one class `B`.
   - The `javadoc` program takes a Java file (e.g., `a.java`) that defines a class and makes an HTML file that describes the class (e.g., `a.html`).
   - You need to add a license agreement to the top of every HTML file that `javadoc` produces. The contents of the license are in a file `license`. You have written a shell-script `add-license` that takes an HTML file and changes it so it includes the contents of `license`.

   (a) **8pts** Write a `Makefile` with targets for making `a.html` and `b.html`. The generated files should include the license. They should be remade whenever and only whenever a file that could affect their contents has changed.

   (b) **4pts** Which of the files mentioned in this problem would you put in a version-control system? Briefly justify your inclusion or exclusion of each file.

   Solution:

   (a)
```
a.html: a.java license add-license
        javadoc a.java
        add-license a.html

b.html: b.java license add-license
        javadoc b.java
        add-license b.html
```

   (b) `a.html` and `b.html` should *not* go in the repository because they are automatically generated. All the other files should: The Java and license files are inputs to make the HTML files. add-license is a program written for this task; its contents affects the result. Also, the `Makefile` should go in the repository so other developers can use it. (No points deducted for not discussing `javadoc`, which should not go in the repository because it is an executable and is a tool used (not developed) by this project.)

Name:____

8. Consider this Java code. Do *not* assume there is only one thread.

```
final class A  { // final means no subclasses, so that is not an issue
  private int i = 0;
  private Object lk;
  public void f() { synchronized (lk) { ++i; ++i; } }
  public boolean g() { synchronized (lk) { return (i % 2)==0; } }
}
```

(a) **2pts** Can a call to `g` ever return `false`? Why or why not?

(b) **2pts** If we change the body of `f` to just `{++i; ++i;}`, can a call to `g` ever return `false`? Why or why not?

(c) **2pts** If we change the body of `g` to just `{ return (i % 2)==0; }`, can a call to `g` ever return `false`? Why or why not?

Solution:

(a) No. Only `f` can change `i` and `i` is always even before and after `f` runs. `g` cannot execute its return statement while `f` is in the middle because both methods acquire the same lock.

(b) Yes, now one thread might run `g` (including acquiring the lock in `lk`) when another thread is running `f` and has incremented `i` exactly once.

(c) Yes, now one thread might run `g` when another thread is running `f` and has (acquired the lock in `lk`) and incremented `i` exactly once.

Solution:
Note: This question has a small bug that nobody discovered: There should be a constructor that initializes `lk` to a new object!