Name:_____

CSE 303, Winter 2006, Final Examination
16 March 2006

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 10:20.**
- You can rip apart the pages, but please write your name on each page.
- There are **100 points** total, distributed **unevenly** among **7** questions (which 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. (**20** points) This problem has 4 parts.

   Suppose you have a large C program named `app` that includes a function `f`. You want to know if running `app` on the input `17` causes `f` to be called. For each question below, be *very specific* about how you would modify files and/or run commands and programs, and in what order.

   (a) Describe how to solve this problem by adding code to a C file.

   (b) Describe how to solve this problem without changing code but using the debugger `gdb`.

   (c) Describe how to solve this problem without changing code but using the profile `gprof`.

   Now suppose `app` also has a function `g` and you want to know if running `app 17` causes `f` to be called after `g` is called but before `g` returns. (That is, does `g` ever cause a sequence of calls that includes `f`?) You may assume `g` is called a small number of times.

   (d) Describe how to extend the *first two* approaches above to solve this modified problem.

   Solution:

   (a) Put a print-statement (e.g., `printf("A")`) at the beginning of the body for `f`. Recompile the program. Run `app 17` and see if the print-statement occurs. (To be super careful, you should write to a different file in case the application prints `"A"` for other reasons, but nobody actually does that.)

   (b) Recompile the program with `-g`. Run `gdb app`. Set a breakpoint at `f` (command `break f`). Run the command `run 17`. If the breakpoint is reached (before the program terminates), `f` is called.

   (c) Recompile the program with `-pg`. Run `app 17`. Run `gprof app` and look at the resulting call-count output to see if `f` has a non-zero count.

   (d) To extend the first approach, also put a print statement (e.g., `print("B")`) at the beginning of `g` and another (e.g., `print("C")`) at the end of `g`. Recompile the program. Run `app 17` and see if (using our examples) there are one or more `A`s in-between a `B` and a `C`.

   To extend the second approach, also put break statements at the beginning and end of `g` before running the command `run 17`. As each break point is reached, use `continue` to proceed. See if the breakpoint for `f` ever occurs in-between the beginning of `g` and the end. (Note: If `f` is called many times not from `g`, then you want to have the breakpoint at `f` only while `g` is running. You can do this by adding the breakpoint when stopped at the beginning of `g` and deleting it when stopped at the end of `f`.)

Name:_____

2. (**20** points) Consider these two C files:

a.c:

```
void f(int p);

int main(int argc, char**argv) {
  f(17);
  return 0;
}
```

b.c:

```
void f(char * p) {
  *p = 'x';
}
```

(a) Why is the program made from `a.c` and `b.c` incorrect? What would you expect running it to do?

(b) Will `gcc -Wall -c a.c` or `gcc -Wall -c b.c` give an error or produce `a.o` and `b.o`?

(c) Will `gcc -Wall a.c b.c` give an error or produce `a.out`?

(d) How would you use a standard C coding practice (using an extra file) to avoid the problem above? Write this extra file and modified versions of `a.c` and `b.c` to explain.

Solution:

(a) `f` expects a `char*` but is passed 17. So it treats an `int` as a pointer, and will probably seg-fault as it tries to write `'x'` to address 17.

(b) Both will work without error; `a.o` will "think" `f` takes an integer.

(c) This will also work; the linker does not have type information, so since `f` exists it will combine `a.o` and `b.o` to produce an executable. (Note with C++-style name-mangling we would get an error, but this question asks about C.)

(d) Use a header for file for prototypes and have files using and defining the functions include the header file. That way `b.c` will not compile if the definition of `f` disagrees with the prototype or `a.c` will not compile if the use does not agree (as shown below).

b.h:

```
#ifndef B_H
#define B_H
void f(char* p);
#endif
```

a.c:

```
#include "b.h"
int main(int argc, char**argv) { f(17);  return 0;}
```

b.c

```
#include "b.h"
void f(char * p) {*p = 'x'; }
```

Name:_____

3. (**15** points) Write a `Makefile` for this scenario:

   - An application `myprog` is written in C, with all the code in `myprog.c`.
   - You wrote two test-inputs, in files `input1` and `input2`.
   - You want to run `myprog` *with profiling* on each test-input and then use `gprof`, saving the result to file `prof1` (for input `input1`) or `prof2` (for input `prof2`).
   - You have a bash script `compare` that takes as arguments two files created by `gprof` and produces an interesting summary. You want a phony `run` target that runs `compare` on `prof1` and `prof2`.

   Your `Makefile` should re-compile or re-run programs only as necessary (except the `run` target should always execute `compare`), but it should never use out-dated programs.

   Hints: You should have 4 targets. Some will need multiple commands. Some will need multiple sources.

   Solution:

```
myprog: myprog.c
        gcc -o myprog myprog.c -pg

prof1: myprog input1
        ./myprog input1
        gprof myprog > prof1

prof2: myprog input2
        ./myprog input2
        gprof myprog > prof2

run: compare prof1 prof2
        compare prof1 prof2
```

Name:_____

4. (**10** points) Suppose you are using `cvs` for a group project. You decide to move some of the code in `foo.c` to a new file `bar.c`. You update the `Makefile` appropriately.

   (a) What `cvs` command should you use before your next commit?

   (b) If you forget to do your answer to part (a), who will discover your forgetfulness and when?

   Solution:

   (a) `cvs add bar.c`

   (b) Another group member will do `cvs update` and then try to build the program. At this point, they will not have all the code so the compiler or linker will give an error.

Name:_____

5. (**8** points)

Suppose you want to make a library `blah` (also known as an archive, i.e., a `libblah.a` file) containing functions `f1`, `f2`, ..., `fn` that may call each other but do not call any other functions.

(a) If you want to make sure library users never have to write `-lblah` more than once when linking, how should you organize your n functions into files? Explain.

(b) If you want to make sure library users never have any more code in their executable than absolutely necessary, how should you organize your n functions into files? Explain.

Solution:

- Put all the functions in one file. That way if any function is used in a program, all the code in the library will be in the executable. So adding a second `-lblah` would never cause any additional code to be included.
- Put each function in a separate file. That way for each occurrence of `-lblah`, the linker will not include code that is neither called from outside the library or from an earlier function in the library that was already included. However, the user may need to include `-lblah` multiple times, depending on the order of the `.o` files in the library and what functions in the library call what other functions.

Name:_____

6. (**12** points) Here is a C program for testing a function `f` to see if it always returns `0`:

```
int f(int x, int y);
int main(int argc, char** argv) {
  if(f(0,0)!=0)
    return 1; // failure
  if(f(1,1)!=0)
    return 1; // failure
  return 0; // success
}
```

Give an example of a function `f` such that:

- The test above achieves full statement and branch coverage.
- The function does *not* always return 0.

Explain your answer.

Solution:
(There are an infinite number of answers.)

```
int f(int x, int y) {
  return x-y;
}
```

In this particularly short answer, any input gets full statement/branch coverage, since there is one statement that always executes. But any input where the arguments are different leads to a non-zero answer, for example `f(0,1)`.

A more expected answer is something like:

```
int f(int x, int y) {
   ans = 1;
   if(x==0) ans = 0;
   if(y==1) ans = 0;
   return ans;
}
```

Here `f(2,2)` gives a non-zero answer. We have full coverage because the first test covers the first if being true and the second false; and the second test covers the first being false and the second being true. So together the tests execute every statement and take every branch.

Name:_____

7. (**15** points) This problem has 3 parts.

Assume we are using reference-counting to manage the memory pointed to by `p` and `q`. Recall that with reference-counting, when we assign `q` to `p` we should write:

```
decr_count(p); // line 1
p = q;
incr_count(p); // line 3
```

(a) What error could occur (later) if you forget line 1? Explain.

(b) What error could occur (later) if you forget line 3? Explain.

Suppose the definition of `incr_count` looks like this:

```
void incr_count(struct Foo * x) {
  int c = x->count;
  x->count = c + 1;
}
```

(c) If two *threads* call `incr_count` with the same pointer at the same time, what could go wrong? What would happen to the count and what error could occur later as a result?

Solution:

(a) A space leak: (Unless `p==q`), the count for the object `p` points to before the assignment is now too high; it is greater than the number of references to it. So we will never reclaim the object.

(b) A dangling-pointer derference: The count for the object `p` points to after the assignment is now too low; we will reclaim the object when there is still 1 reference to it. If that reference is later used, the computer may catch fire.

(c) There is a race condition. For example, if one thread executes the first line, then the other thread executes both lines (for the same pointer), then the first thread executes the second line, then the count will be increased by 1 instead of 2. So the count will be too low, potentially leading to a dangling-pointer dereference as in part (b).