CS 240 (Park) **Midterm 1** **June 30 (Thu.), 2016**

Remarks: Keep the answers compact, yet precise and to-the-point. Long-winded answers that do not address the key points are of limited value. Binary answers that give little indication of understanding are no good either. Time is not meant to be plentiful. Make sure not to get bogged down on a single problem.

PROBLEM 1 (36 pts)

(a) What happens when we run gcc with options -E and -P? What about running gcc with option -c? What is static linking? What transpires when we try to "run" an executable file a.out at a shell prompt (i.e., % a.out)?

(b) Suppose we initialize and print two string variables r and s as follows:

> char r[4], s[4]; r[0] = 'A'; r[1] = '\0'; r[2] = 'C'; r[3] = 'D'; s[0] = 'a'; s[1] = 'b'; s[2] = 'c'; s[3] = 'd'; printf("%s %s", r, s);

What gets printed and why?

(c) Given the following code snippet

> int x, *y, *z; x = 10; y = &x; *y = 20; *z = 30; printf("%d %d %d", x, *y, *z);

what is likely to happen when it is compiled and executed? What is likely to happen if the statement *z = 30 were changed to z = 30 while keeping all else the same?

PROBLEM 2 (34 pts)

(a) Describe how a 1-dimensional array, int d[4], whose members are set to d[0] = 0; d[1] = 1; d[2] = 2; d[3] = 3; is laid out in memory. Use a drawing with example addresses. How can the statement d[2] = 2 be rewritten so that square brackets are not used? Explain what the revised statement does.

(b) An important component of the run-time environment of a C program is the stack memory (i.e., "scratch space") allocated to a function when it is called. What are the three main uses of stack memory? Suppose we have a function

> void myfunc(void) { int i, m[3]; for (i = 0; i < 10; i++) m[i] = i; }

that is called from main() in a C app. What will happen and why, assuming we used gcc in our lab to compile the app? If we were to compile with gcc option -fno-stack-protector, what is likely to happen? Explain.

PROBLEM 3 (30 pts)

(a) Suppose we declare a 2-dimensional array, char u[2][3], and perform the assignment

> u[0][0] = 'g'; u[0][1] = 'o'; u[0][2] = '\0'; u[1][0] = 'P'; u[1][1] = 'U'; u[1][2] = '\0';

How can the statement u[1][2] = '\0' be rewritten so that it does not use square brackets? Explain what the revised statement does. In Problem 1(b), we printed a string (i.e., 1-D character array char r[4]) by calling printf() with format %s and argument r. Keeping in mind that r contained an address (i.e., r is a pointer), how would we call printf() with format %s to print the string "PU" stored in the second row of the 2-D array char u[2][3]? Explain why what you are doing is (or should be) correct.

(b) Based on our discussion of addresses and pointer variables, what is the meaning of a variable d that is declared as int ***d? Based on the material covered in class so far, write a short code snippet that ends with the two statements

> ***d = 7; printf("%d", ***d);

that compiles, does not crash, and prints the value 7.

BONUS PROBLEM (10 pts)

Based on our discussion of how memory for local and global variables in C are allocated, what is their difference? Assume float z is a local variable of some function somefunc(). Suppose main() calls somefunc() and somefunc() returns to main() after executing its code. What happens to z when somefunc() returns? What value will z contain when somefunc() is called a second time by main()?