Problems [215 pts]

Problem 1 (20 pts)

Q. Compile the program in v1/ once to generate the executable file a.out once, but run a.out four times. Specify what output you observe and explain why some of it changes across different executions.

A. When I run the a.out, it prints two things. One is `7` which is the value of `s` and the address of the variable s. However, every time I runt the a.out, it gives me different values. %p is a format specifier for pointer type. Every time I run a.out, the executable program stores the variable in the different address. This is the reason why it shows the different outputs.

Q. What happens to the output if the format of the second printf() call is changed from %p to %d?

A. When I compile, the compiler gives me the following warning.

main.c:14:12: warning: format ‘%d’ expects argument of type ‘int’, but argument 2 has type ‘int *’ [-Wformat=]

```
  printf("%d\n",&s);
```

```
       ~^    ~~
```

```
       %ls
```

And the following is the output.

`-1384103180`. Every time I run the a,out, it gives me different values which means that this value is a garbage value.

If I use `%d` to print the address of the variable s, the compiler cast the address of the variable into a signed integer type.

Q. What happens if the format is changed to %u? Explain your finding.

A. The compiler gives me such warning when I compile the main.c file.

main.c:14:12: warning: format ‘%u’ expects argument of type ‘unsigned int’, but argument 2 has type ‘int *’ [-Wformat=]

```
printf("%u\n",&s);

        ~^    ~~

        %ls
```

And the following is the output.

`101213316`. Every time I run a.out, this gives me different values which mean that those values are the garbage values.

If I use `%u` to print the address of the variable, the compiler casts the address to the unsigned integer which only shows a positive integer.

Problem 2 (20 pts)

Q. Explain why after calling changev1() in v2/ the value of x printed on stdout remains 7 but changes to 2 after calling changev2().

A. The line on where the `changev1()` is called, the function is passing the integer type. What the `changev1()` function is doing is setting the passing variable as `2`. But this function will only change the value of `b` itself. Not the actual value of variable x, because the function is not passing the address of the `x`.

However, the function `changev2()` is passing the `&x` which means this address of x. By accessing the address of x, the function can change the value of x to 2 where the 7 is stored.

Problem 3 (20 pts)

Q. Code a third function, void changev3(void), in v2/main.c so that after main() calls changev3() and prints x to stdout its value will have changed to 5. Explain why the third method works.

A. I removed the local variable initialization of x and made a global variable out of the main function. The reason why it works is that `void changev3(void)` is passing nothing as a return type. Therefore, I can just change the value of x in the `void changev3(void)` function. In the function, it changes the value of x directly on the line of `x = 5;` and this finally prints the x to 5.

Problem 4 (15 pts)

Q. Create a copy of main.c in v2/ and name the file main1.c. Modify changev2() in main1.c so that the assignment statement, *b = 2, is changed to b = 2. Compile and execute a.out. Explain why x in main() remains 7 after calling changev2().

A. The `changev2(int *b)` function is passing a pointer type. Therefore, unless I change the address of b to two, then the value of x will not be changed. However, `changev2(int *b)` in main1.c is changing the value of b to 2 which is just changing the value of stack's `b` and does not point to the actual value of x. Therefore, the x remains 7.

Problem 5 (15 pts)

Q. Modify the code in v3/ so that the function modv() is placed in a separate file aaa.c in v3/ and the function declaration, void modv(int *), along with the C preprocessor directive, #include , are put in a separate file bbb.h in v3/. Compile and run a.out to test that the modified implementation works correctly.

Finished

Problem 6 (10 pts)

Q. Create a new subdirectory v3a/ under lab2/, then copy all files of v3/ to v3a/. Inside v3a/, create object files of the two .c files by running gcc with option -c. Verify that the respective objective files have been created. Link the two .o files by running gcc and creating an executable file named, main.bin, by using the -o option. Verify that the main.bin runs correctly.

Finished

Problem 7 (20 pts)

Q. Modify main.c in v4/ by adding the assignment statement, c = &b, at the end. Explain how the variable c must be declared in main.c for the assignment to make sense. Make a printf() call after the assignment statement to output the value of variable, a, using the variable c. Compile main.c and test that your code changes work correctly.

A. Variable `c` must be declared as `int **c`. This is because in the statement that I add, `c = &b`, there is an ampersand in front of the `b` even though `b` is already the pointer type. Therefore, c should dereference `&b` for two times to get the actual value by putting two asterisks in front of the `c` when I declare it.

Problem 8 (20 pts)

Q. Modify main.c in v8/ by introducing a new variable, int def, and adding the assignment statement, abc = &def, before the assignment statement *abc = 100. Compile and verify that running a.out generates a segmentation fault. Using the debugging technique discussed in v6/ that makes use of conditional C preprocessor directives, add debugging code to main.c in v8/ to pinpoint which statement causes the segmentation fault. Explain why the segmentation fault occurs where it does.

A. In the code, on `abc = &def;` this statement, abc has the same address as the `def` and only the memory for `def` is allocated. However, `*(abc+2) = 300;`, this line is causing the segmentation fault. The reason why this line causes the segmentation fault is that even thought the memory for `(abc+2)` is not allocated, the program is trying to access the memory which is illegal. Therefore, the segmentation fault occurs.

Problem 9 (20 pts)

Q. Explain what the silent run-time error in the example code of v9/ is. What negative consequence can it have?

A. Since abc has 5 spaces, in the for loop, 5 is the biggest number that the k can be counted. If k is counted more than five times, then it should cause the runtime error. However, when I do `gcc main.c` and run `a.out`, it does not give me any error. The compiler is supposed to give me the runtime error, but it is not giving me any error. This is called silent run-time error.

The negative consequence this can cause is even though the program is not giving me an error and I don't even know it has the error, the program will cause a big error when the program got bigger as the silent run-time error is stacked over time.

Q. Make a copy of main.c in v9/ and call it main1.c. What happens if you change the limit of the third for-loop from 6 to 7 in main1.c?

A. If I change 6 to 7, the compiler says `*** stack smashing detected ***: <unknown> terminated`. Even though the compiler gives me such error, the program is still running.

Q. What happens if you instruct gcc not to perform stack overflow detection by running gcc with option -fno-stack-protector?

A. If I run the program with the command, it is not showing the error message like `*** stack smashing detected ***: <unknown> terminated`.

Q. What happens if you keep increasing the third for-loop limit to 8 and above?

A. If I increase the for-loop limit to 100, it just gives me the following message. `*** stack smashing detected ***: <unknown> terminated`.

However, if I increase the limit to 10000, then it gives me the `segmentation fault` and it does not run the code after the line where causes the segmentation fault.

Problem 10 (20 pts)

Q. In CS240, unless otherwise specified, utilize gcc's help of inserting code that helps detect stack overflow at run-time. Make a copy of main.c in v9/ and call it main2.c. Modify main2.c by changing the limit of the third for-loop in main.c from 6 to 7 and declaring the array int a[5] as global. Even though gcc's stack overflow detection is enabled, what happens when you compile main2.c and execute a.out?

A. If I set `abc[5]` as a global variable and execute the a.out file, the program is running without any problem even the message of `stack smashing detected`.

Q. Can you explain why the behavior is different from that of Problem 9?

A. If I declare `int abc[5]` in the main() function which is a local variable, the memory will be stored in the stack memory. However, if I declare `int abc[5]` outside of the main() function, it becomes the global variable and stored outside of the stack. The thing is gcc only checks the stack but `*** stack smashing detected ***: <unknown> terminated`, this message is given by the gcc. If the variable is located outside of the stack, the message will not be given, and this is the reason why problem 9 and problem 10 shows the different behavior.

Q. What happens when you increase the loop limit of the third and fourth for-loops from 7 to 1000?

A. The program only gives me the following: *** stack smashing detected ***: <unknown> terminated if I increase the limit from 7 to 1000. Moreover, the program prints the output properly.

Q. What about 1020? Through trial-and-error identify the loop limit at which the program crashes.

A. If I set the for-loop limit as 2089, it gives me the following: *** stack smashing detected ***: <unknown> terminated.

However, if I set the limit as 2090, it crashes and shows me `segmentation fault`.

So, the limit is 2089.

Problem 11 (15 pts)

Q. What is the logical layout of the 1-D character array, *u*, in v10/main.c in main memory as discussed in class? Do not confuse the logical layout from the physical layout which is a by-product of optimized memory usage by C's compiler. The latter is of marginal relevance when understanding how to program with pointers in C.

A. `char u[7]` is an array that has 7 char elements in it. Since `u` is a pointer type of a character, `u` is pointing to the address where the actual char element is in the memory. Therefore, if I want to read the second element of the array, we can do `*(u+1)` since it is a pointer. Then this will point to the `U`. The compiler knows `+1` means 1 byte because the type of the element is character.

Q. Replace the fourth character of u[3], 'D', with the character '\0' and output the modified u to stdout. What do you observe and why?

A. The result is `PUR`. The reason is that a character `\0` means the end of the string to the compiler. Therefore, when `\0` comes out as an element, the compiler will stop reading the array and will print the elements before the first `\0` which is `u[3]`, `PUR`.

Problem 12 (20 pts)

Q. The library function, strlen(), returns the length of its input which is supposed to be string. In your own words or pseudo-code, describe the logic behind strlen() which is but a few lines of code.

A. Strlen() is passing the string or a character pointer as an argument. I can run the while loop with this function. Since every character array, which is string, has `\0` at the end, run the while loop until they find the `\0`. Every time the while loop runs, I count the numbers. Also, I add 1 to the pointer the function is passing, because in this way, the loop can go to the next element of the array. When the loop meets `\0`, stop counting and return the number counted. The returning number will be the length of the string.

Q. Suppose a programmer calls strlen() but provides as input a 1-D character array that is not a string. What can happen as a result of calling strlen()? What should

happen when strlen() is called where the input is not a string? Discuss your reasoning.@

A. I tested three cases of putting char array in ‘strlen()’.

If there is a null character, `\0`, among the elements, it only reads until right before the `\0`.
If there are the proper numbers of characters as the number of the size of the array, then it prints the correct length of the array, because there is the null character, `\0`, at the end by default.
If the number of characters exceeds the number of elements of an array, then it prints some random values after the last element.

If strlen() is passing the argument which doesn’t have `\0` at the end of the array, I think the function should return the error. This is because character array without `\0` at the end of the array mean that is not a string and strlen() is the function that return the length of the string, not the character array.

Bonus problem (20 pts)

Q. The silent run-time bug brought forth by Problems 9 and 10 may, in part, be mitigated through improved software engineering practice. Instead of specifying array sizes and limits of for-loops by numerical constants, the C preprocessor directive #define is used to define a macro that is used in place of numerical constants. Make a copy of main.c in v9/ and call it main3.c. Use the macro MAXARRAYSIZE defined as 5 to modify main3.c so that silent run-time bugs from array overruns are prevented. Compile and test that your code works correctly.

Finished.

The Bonus Problem is entirely optional. Bonus points count toward reaching 45% of the course grade contributed by lab assignments.