Problem 1 (20 pts)

1. **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.**

   Every time I run a.out, it prints out 7 as the value of int s, but the address of int s that gets printed out seems to change each time. This is because the computer is running other processes that use memory at the same time, so each time a.out is run it uses the first block of free memory that it finds, which may change as the memory used by other processes also changes.

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

   I get a warning:

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

   However, a.out is still changed and now prints an decimal value of the address rather than a hexadecimal value starting with 0x. The new values printed are 9 digits and can be negative or positive. This is because %d specifies an int, so the address that is usually conveyed as a hex value is converted from a hex value to a decimal value. Sometimes it appears to be negative because of 2’s complement, but realistically memory doesn’t have negative values. This address still changes every time a.out is run.

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

   I also get a warning here:

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

   However, a.out is still changed and now prints a decimal value of the address rather than a hexadecimal value. The new values here are 10 digits and always positive. This is because %u specifies an unsigned int, so the address that is usually conveyed as a hex value is converted from a hex value to a decimal value, not using 2’s complement in this case, which is why there are no negative values. The address still changes every time a.out is run.

Problem 2 (20 pts)

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

The value of x isn't changed from its original value of 7 by changev1() because changev1() doesn't change the value at the address of x. It is ineffective because simply using the name of the variable doesn't convey to the computer that you are trying to change the same variable in the main method. It only understands changev2(), which utilizes the address of the variable to identify what needs to be changed across functions.

Problem 3 (20 pts)

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.

In order to make it work, I had to declare x as a global variable so that changev3 could access it. Then instead of using the variable b that changev1() and changev2() use, I wrote x = 5. This way, I accessed the global variable x and changed it even though I passed in a void value. However, changev1 and changev2 still work (or don't work, in the case of changev1()) the same way they did before because of the way they are programmed to pass in the value of x in the main method.

Problem 4 (15 pts)

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().

First, I get a warning:
main1.c:37:5: warning: assignment makes pointer from integer without a cast [-Wint-conversion]

x remains unchanged because the assignment statement doesn't attempt to change the value at the address once the pointer is removed. Even though an address is passed in, without using a pointer it is not properly accessed by the assignment statement and nothing is changed.

Problem 5 (15 pts)

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.

Problem 6 (10 pts)

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.

I used:

gcc -c main.c

gcc -c aaa.c

gcc main.o aaa.o -o main.bin

Everything seems to work correctly.

Problem 7 (20 pts)

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.

c must be declared as **c because it points to b, which is already a pointer declared as *b. Therefore it is a pointer to a pointer.

Problem 8 (20 pts)

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.

The debugging statements indicate that the problematic statement is “*(abc+2) = 300;”.

This is because the +2 navigates out of the stack of memory allocated to main.c. Since there are only two variables to begin with, it is only allocated two blocks of memory, and once you try to access a third block the program fails.

Problem 9 (20 pts)

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

   The program is writing 5 to a memory slot that doesn't belong to abc because abc is initialized only to 5 spots--0 through 5 is six numbers. It didn't crash because that slot belongs to other parts of the program, but it may have erased another value that was being stored in the memory and may cause harder to detect and more errors down the line.

2. **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?**

   It compiles and produces a.out. Then a.out prints out the numbers as usual but after reaching 5 in the second printing loop, you get an error:

   *** stack smashing detected ***: <unknown> terminated

   Aborted

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

   It compiles and produces a.out. When a.out is run, it prints numbers up to 5 in the second printing loop, and then ends. It does not print the stack smashing/aborted error.

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

   It has the same results as when the limit is changed to 7, both compiling normally and with -fno-stack-protector.

Problem 10 (20 pts)

1. **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?**

   It acts as if the stack overflow detecting was turned off. It compiles and produces a.out. However when run, it prints the numbers up to 5 in the second loop and then ends without printing any errors.

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

For a program like a.out, a certain amount of memory is allocated. The memory for functions is on one end and the memory for includes and global variables is on the other side. There is only a certain amount of memory allocated for the main() function and the variables within it, so when a[] is declared in main, the computer will not allow its size to surpass what was declared. However, when a[] is declared as a global variable, there is some “buffer” space between the memory allocated for globals/includes and functions, and if you try to add more indices to a[] than originally declared, then it will start running into this in-between “buffer” memory. When a[] is a global variable, the program will only produce a segmentation fault once you expand a[] to the point that it starts to run into the memory allocated for the main() function.

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

   It prints up through number 999 and ends.

4. **What about 1020?**

   It prints the first loop (up to 4), but none of the second loop and produces a segmentation fault.

5. **Through trial-and-error identify the loop limit at which the program crashes.**

   It crashes at loop limit 1017.

Problem 11 (15 pts)

1. **What is the logical layout of the 1-D character array, *u*, in v10/main.c in main memory as discussed in class?**

   The logical layout of a 1D character array can be a pointer to a section of memory, which you could draw as a rectangle subdivided into a sequence of the same number rectangles as number of characters in the array.

2. **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. 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?**

   It simply says PUR. This is because \0 is a null terminator and indicates the end of the string. Because of the placement at D, printf thought that the full string to print was just PUR.

Problem 12 (20 pts)

1. **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.**

   strlen() takes in a character array and counts the number of characters up until a null terminator, not including the null terminator. Then, it returns that number.

2. **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()?**

   I made a test program test.c and tested it with a short character array at first. I encountered no errors, but when I made it a very large character array (size 100,000) I encountered a silent runtime error where it thought that the total length was 100,006. It would also be possible to encounter segmentation faults or stack smashing errors. These problems happen because strlen() is not meant to be used on character arrays that don't end with null terminators.

3. **What should happen when strlen() is called where the input is not a string? Discuss your reasoning.**

   It should produce an error indicating the use of an incorrect type of variable. When I tested this myself with an int array, the error message was:

   test.c:7:20: warning: passing argument 1 of 'strlen' from incompatible pointer type [-Wincompatible-pointer-types]

   In file included from test.c:1:0:

   /usr/include/string.h:384:15: note: expected 'const char *' but argument is of type 'int *'

Bonus problem (20 pts)

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.