Problem 1 (20 pts)

We discussed the code in v1 and verified that it works correctly as intended: the integer read from keyboard input overwrites the previous value of s which is output on the terminal display. Print what is actually being passed to scanf() when the second argument is &s. Why does changing scanf("%d",&s) to

scanf("%d",s)

generate a segmentation fault run-time error?
By removing the &s and replacing it with s, you are effectively telling the scanf() function to place the scanned data in an area with the address '8', the initial value assigned to *s*. Doing this causes a segmentation fault error due to the fact that '8' is not a valid address in memory.
What is the meaning of segmentation fault?
A segmentation fault occurs when the program attempts to access a section of memory that it does not have access to, or in this case, does not exist.

Problem 2 (20 pts)

The code in v1 declares s as an integer variable. Leaving the declaration of s as is, define a new variable u that is declared as an integer pointer, i.e.,

int *u;

Rewrite the code so that u is used in place of s, i.e., as u, *u or &u depending on which usage is correct. Explain in lab2.pdf why your way of coding is correct.
Due to the fact that the variable was declared as *u, the scan line replaced &s with u, as u refers directly to the address. printf() requires the actual value, so in that we use *u, which has the actual data we want to access. Finally, the print line is simply &u, due to the fact that it is explicitly stated (%p) as referring to an address.
When submitting your modified code, v1/ should contain just the revised code main.c. Put lab2.pdf under lab2/.

Problem 3 (20 pts)

Similar to Problem 2, declare s as a float pointer and rewrite the code in v2/ so that it works correctly. Also, put the function changeval() in a separate file changeval.c. Note that the function definition (i.e., prototype) of changeval() needs to be changed. When compiling the two C source files main.c and changeval.c, generate separate object files by using the -c option in gcc. Then run gcc on the two object files to generate executable machine code. When submitting your work, v2/ should contain the source files, object files, and executable binary.

Problem 4 (30 pts)

Consider the code in v8/ which is similar to the code in v7. What happens when you compile and run the code? Using the basic debugging technique discussed in class for the example code in v6/, track down which part (i.e., statement) of the code triggers the run-time error.
***h = 100;**
Explain what is going on and why the code in v8/ differs fundamentally from the one in v7.
A value (100) is being assigned to the pointer *h, which hasn't actually been defined off a specific location in memory.
How can you fix the code?
If int*h is compared to another variable rather than assigned to nothing, this error can be avoided. For example:
int h*;
int x;
x = 100;
h = &x;
The general idea is repeated with the other variables.

Problem 5 (30 pts)

The code in v9/ contains a silent run-time bug. Explain what that is.
A silent runtime bug is as it's name implies, a bug the occurs during runtime that does not interrupt the flow of the program (via a crash or what have you), nor throw any error messages.
What happens when you change the loop bound from 6 to 7? Explain what is going on.
The loop goes beyond the boundaries of the array, and subsequently crashes the program.
What is one programming method for preventing the above run-time bugs?
Careful and diligent programming, and in the event that one of these bugs occurs, careful debugging.

Problem 6 (80 pts)

Write a C function, readinput(), with the following function definition (i.e., prototype)

int readinput(char *);

where the caller, in our case main() in v10/, passes a pointer to a string declared in main()

char c[10];

The function readinput() reads from stdin (i.e., keyboard) a string that is terminated by pressing the RETURN/ENTER key ('\n') on the keyboard. The valid string inputs are: "ne" and "oxi". If "ne" is input, readinput() returns 0. If "oxi" is input, it returns 1. For all other input, readinput()

returns -1. Check how main.c in v10/ calls readinput() and uses its return value. An object code readinput.o from a sample program (but not its source code) has been placed in v10/ for testing purposes.

Put the code of readinput() in its own file, readinput.c. Compile main.c and readinput.c separately with option -c. Link them by running gcc on their object files main.o and readinput.o. Test the executable on valid ("ne" and "oxi") and invalid (all else) input to verify that it works correctly. When submitting your code, v10/ should contain main.c and readinput.c. Provide adequate comments in readinput.c.

Important: Do not use string processing library functions to carry out string processing tasks in readinput(). Write your own code that performs the string comparisons with "ne" and "oxi" using if-else (character by character). Use the scanf() to read the string from stdin.

Finished.

Bonus Problem (30 pts)

An extension of Problem 6, what happens when you input a long string, say, "mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm" or something similar as input when the app runs? Why does what you observe happen?
An error is passed, due to the inputted string being beyond the length of the char array used to contain it.
Fix the problem by using getchar() to read the string in readinput() instead of scanf(). When a user enters input that is longer than 9 characters, print a special message. Create a directory v11/ and put your modified main.c and readinput.c code in v11/.

Not attempted.