CS240 Midterm Solution, summer 2022

P1(a) 12 pts

The first printf() outputs 3 since b is a pointer to variable a.
4 pts

*c = 5 is likely to generate segmentation fault since the code does not place a valid address in c.
5 pts

The second printf() is likely not reached due to segmentation fault from *c = 5 which terminates the running program.
3 pts

P1(b) 12 pts

Outcome 1: prints "Hi" to stdout.
4 pts

Outcome 2: prints "Hi" followed by additional byte values.
4 pts

If the memory location r[2] contains EOS then the first outcome results. Otherwise, printf() will print continue to print byte values (not necessarily ASCII) until a byte containing 0 (i.e., EOS) is reached.
4 pts

P1(c) 12 pts

Equivalent to x[1][2].
6 pts

In our logical view of 2-D arrays: x points to the location in memory where the beginning addresses of two 1-D integer arrays are located. Therefore x+1 points to the beginning address of the second 1-D integer array. *(x+1) follows the pointer to the beginning address of the second 1-D integer array.
3 pts

*(x+1)+2 results in the address at which the third element of the second 1-D integer array is stored. *(*(x+1)+2) access the content of the third element of the second 1-D integer array. Hence equivalent to x[1][2].
3 pts

P2(a) 16 pts

First call: returns 4.
3 pts

This is so since if-statement checks 4 > 2. a++ returns 4 before incrementing a.
3 pts

Second call: returns 4.
3 pts

If-statement checks 4 > 3 since static int b preserves its previous value. Hence a++ returns 4.
3 pts

Third call: returns 5.
2 pts

If-statement checks 4 > 4 since b is static. The else-part increments b first before returning b, hence 5.
2 pts

P2(b) 16 pts

Since we did not assign a valid address to n, **n is likely to reference an invalid address that triggers segmentation fault which terminates the running program.
5 pts

Although the first printf() call was successful, 3.3 will likely will not be output to stdout (i.e., display) due to abnormal termination of the program and buffering by stdio library functions.
3 pts

Adding newline in the first printf() call, or calling fflush(stdout) after the first printf() call will force 3.3 in the stdout buffer to be flushed before the program terminates due to segmentation fault.
8 pts

P3(a) 16 pts

scanf() does not prevent user input that exceeds 100 characters from overwriting memory in readpasswd()'s stack frame, potentially modifying its return address. This can lead to execution of unintended code such as malware.
8 pts

Code should explicitly check that no more than 100 characters are read from stdin to prevent overflow over secret[100]. This can be done by reading character by character using getchar() in a loop until newline is encountered or 100 characters have been read.
8 pts

P3(b) 16 pts

```
#include <stdio.h>

int main()
{
FILE *fp;
int c, count;

  // open input file to read: comments not needed
  if((fp = fopen("test.out","r")) == NULL) {
        fprintf(stderr,"opening file blog.dat failed\n");
        exit(1);
  }
  // 6 pts

  count = 0; // ASCII character counter
  while((c = fgetc(fp)) != EOF) { // for each character in the input file
  // 5 pts

            if(0 <= c <= 127) // it's an ASCII character
              count++;
            // 5 pts
  }

  printf("count = %d\n", count); //output result
  fclose(fp); // not needed since file will close at the end of the program
}
```

Bonus 10 pts

Outcome 1: The for-loop overwrites global memory following s[5] which may, or may not, corrupt program data and computation but does not crash the running program (i.e., silent run-time bug).
5 pts

Outcome 2: The for-loop overwrites global memory following s[5] which exceeds the running program's valid memory, resulting in segmentation fault.
5 pts