Name: _____

1. (10 points) For the following, Check T if the statement is true, or F if the statement is false.

   (a) [X] T [ ] F : In the declaration **char** * name="Joe"; the C compiler reserves 8 bytes for a pointer called **name**, and initializes that pointer to the location in memory where the literal string "Joe" resides.

   (b) [ ] T [X] F : The expression **float** y = 'T' / 100.0; fails because C does not know how to convert a character to a floating point number.
   A character literal is really an ASCII 8 bit integer value, and C has no problem converting that value to floating point before doing the division.

   (c) [ ] T [X] F : The size of a pointer depends on what kind of data it points at.
   A pointer is always 8 bytes long on a 64-bit machine, no matter what kind of data it is pointing at.

   (d) [X] T [ ] F : C protects the value of a caller's variable by copying that variable's value into a parameter when invoking a function. That way, the called function cannot modify the variable. However, we can override this protection by passing in a reference to the variable as an argument, instead of passing in the value of the variable.

   (e) [ ] T [X] F : When you invoke a function in C, C evaluates the argument expressions in the order they appear in your invocation.
   The C standard does not require that argument expression get evaluated in order. In fact, the gcc compiler evaluates the last argument first.

   (f) [ ] T [X] F : When you define a function, you must specify the return type for all functions that return a value. If the function does not return a value, no return type is required.
   Functions that do not return values MUST specify a return type of **void**.

   (g) [ ] T [X] F : If you see an expression in C like **char** c=argv[2][0]; then you can assume that the argv matrix is in row major order.
   Actually, argv is not a two-dimensionial matrix, but a vector of pointers, which is NOT layed out in row major order.

   (h) [X] T [ ] F : The first computer bug was a moth stuck in a relay discovered by a woman who was a naval officer and one of the first computer programmers in the mid-twentieth century.

   (i) [ ] T [X] F : If a C variable is declared using **float** x=7.2; then the statments **int** *y=&x; printf("%f",(*y)); will print 7.2.
   The C compiler issues warning messages because you first set an integer pointer to the address of a float, and secondly try to print a float from an argument which is an int. The assignment to y "works", but *y is the integer interpretation of the floating point bits in x, which will not print as 3.7!

   (j) [ ] T [X] F : When converting from an integer to a floating point number, the floating point version of the number is always more precise because it has more decimal places.
   Integers always represent a precise integer value. Floating point number are almost always an approximation, and less precise.

2. (10 points) Given the following C code:

```
#include <stdio.h>
int fib(int x);
int main() {
    printf("fib(5)=%d\n",fib(5));
    return 0;
}
int fib(int x) {
    if (x<2) return 1;
    return fib(x-1)+fib(x-2);
}
```

What will get printed?

☐ fib(5)=1 ☐ fib(5)=2 ☐ fib(5)=4 ☐ fib(5)=5 ☒ fib(5)=8 ☐ fib(5)=13

Working backwards, fib(0)=fib(1)=1, fib(2)=1+1=2, fib(3)=2+1=3, fib(4)=3+2=5, fib(5)=5+3=8.

3. (10 points) Given the following C program:

Listing 1: cmdf.c

```
#include <stdio.h>
int main(int argc, char **argv) {
    char str[5];
    for(int i=1; i<argc; i++) {
        str[i-1]=argv[i][0];
    }
    str[4]=0x00;
    printf("str is %s\n",str);
    return 0;
}
```

If I compile and run this program using the command: ./cmdf testing 123, I get the following results: str is t1^W. Clearly, there is a bug in the code. What is the most likely explanation for these results?

☐ The C compiler did not reserved enough space for the str array.

☐ The program wrote past the bounds of the str array.

☒ No value was ever assigned to str[2] and str[3].

☐ The str string does not have a null terminator

☐ The value of argv[3][0] and argv[4][0] are undefined.

In the given invocation, argc=3, the loops runs for i=1 which writes to str[0], and i=2 which writes to str[1]. The null terminator is put at str[4], but nothing is written to str[2] and str[3] and since str is uninitialized, we don't know what values will be in those elements.

4. (10 points) The following code is taken from the project installment 2 and 3 printCircuit.c program in the printInstance function:

```
printf("\t%s %s(",instanceModule(in),instanceName(in));
for(int pi=0; pi<instanceNumPins(in); pi++) {
    int pn=instancePin(in,pi);
    int nn=pinNet(pn);
    printf("%s",netName(nn));
    if (pi<(instanceNumPins(in)-1)) printf(",");
}
printf(");\n");
```

The last line of the for loop prints a comma, but does not do so every time. Why is this?

- [ ] The instance pins go from 0 to instanceNumPins(in)-1, so this prints a comma for each instance pin.
- [X] A comma goes after ever instance pin except the last instance pin.
- [ ] Different instances may have different numbers of instance pins.
- [ ] There is a bug in the code... a comma should be printed in every iteration of the loop.
- [ ] If a print circuit has too many commas, it is likely to short-circuit and burn.

Answer the following questions by filling in the blanks.

5. (10 points) Given the following C code:

```
#include <stdio.h>
int countWords(char txt[]);
int main() {
    char str[]="This is  a CS-211 test.";
    printf("There are %d words in: %s\n",countWords(str),str);
    return 0;
}
int countWords(char *txt) {
    int nw=0;
    while(*txt!=0x00) {
        while(*txt!=' ' && *txt!=0x00) txt++;
        nw++;
        while(*txt==' ' && *txt!=0x00) txt++;
    }
    return nw;
}
```

What will this code print if compiled and executed?

There are 5 words in: This is a CS-211 test

6. (10 points) Given the following C code:

```
#include <stdio.h>
int sum(int a, int b) { return a+b; }
int mul(int a, int b) { return a*b; }
int div(int a, int b) { return a/b; }
int main() {
    printf("Result=%d\n",div(mul(sum(1,mul(2,3)),2),sum(div(1,4),div(5,2))));
    return 0;
}
```

What will this code print if compiled and executed?

Result=7

Inside out:
mul(2,3)=6;
sum(1,mul(2,3)) = sum(1,6) = 7;
mul(sum(1,mul(2,3)),2) = mul(7,2) = 14;
div(1,4) = 0;
div(5,2) = 2;
sum(div(1,4),div(5,2)) = sum(0,2) = 2;
div(mul(sum(1,mul(2,3)),2),sum(div(1,4),div(5,2))) = div(14,2) = 7;

7. Given the following C program:

Listing 2: factors.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void factors(int n,int f, int *nf,int facts[]);
int main(int argc, char **argv) {
   int facts[100]; int numfacts=0; int n=atoi(argv[1]);
   factors(n,2,&numfacts,facts);
   printf("Factors of %d: ",n);
   for(int i=0;i<numfacts;i++) { printf("%d ",facts[i]); }
   printf("\n");
}
void factors(int n,int f, int *nf,int facts[]) {
   while (0==n%f) { n=n/f; facts[*nf]=f; (*nf)++; }
   if (n==1) return;
   if (f>=n/2) { facts[*nf]=n; (*nf)++; return; }
   factors(n,f+1,nf,facts);
}
```

(a) (5 points) If the factors program were invoked with ./factors 60, what would the function print?

Factors of 60: 2 2 3 5

(b) (5 points) The numfacts variable is a local variable in the main function, initialized to zero. How can it havea non-zero value after invoking the factors function?

Because we pass in a reference to numfacts, the factors function updates the memory where numfacts value resides.

(c) (5 points (bonus)) When factors invokes itself recursively, why does it use nf as an argument instead of &nf?

Because nf is already a pointer to an integer.

8. Given the following C program:

Listing 3: encode.c

```
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "helpers.h"
void encode(char *in,char *pw);

int main(int argc, char **argv) {
    assert(argc==3);
    char buf[100];
    strcpy(buf,argv[2]);
    printf("Raw text: %s\n",buf);
    encode(buf,argv[1]);
    printf("Encoded text: %s\n",buf);
    return 0;
}

void encode(char *in,char *pw) {
    for(int i=0;in[i]!=0x00; i++) {
        if (in[i]<'a' || in[i]>'z') continue;
        in[i]=n2c(c2n(in[i])+pwNum(pw));
    }
}
```

Which uses the following helpers.h include file:

Listing 4: helpers.h

```
int c2n(char c) {
    if (c<'a' || c>'z') return 0;
    return c-'a';
}

char n2c(int n) {
    if (n<0) n+=26;
    return 'a'+(n%26);
}

int pwNum(char *pw) {
    static int pwi=0;
    int n=c2n(pw[pwi]);
    pwi++; if (pw[pwi]==0x00) pwi=0;
    return n;
}
```

If the decode program is compiled and run, an example output from looks like:

```
>./encode password "c code is more fun"
Raw text: c code is more fun
Encoded text: r cgva wj pdrw xqb
```

(a) (10 points) If the encode program were invoked with ./encode ab testing, what would the encoded result be?

tfsuiog

(b) (20 points) Given the following incomplete code for a decode program:

Listing 5: decode.c

```
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "helpers.h"
void decode(char *in,char *pw);

int main(int argc, char **argv) {
    assert(argc==3);
    char buf[100];
    strcpy(buf,argv[2]);
    printf("Encoded text: %s\n",buf);
    decode(buf,argv[1]);
    printf("Decoded text: %s\n",buf);
    return 0;
}
```

Write the C function to decode the encoded text produced by the encode program above, assuming the same password is provided for both encode and decode. The declaration for the decode function is already provided for you.

```
void decode(char *in,char *pw) {
    for(int i=0;in[i]!=0x00; i++) {
        if (in[i]<'a' || in[i]>'z') continue;
        in[i]=n2c(c2n(in[i])-pwNum(pw));
    }
}
```

Question:	1	2	3	4	5	6	7	8	Total
Points:	10	10	10	10	10	10	10	30	100
Bonus Points:	0	0	0	0	0	0	5	0	5