Q1

The output would be:

```
x=_, y=_
 y+2 = c
```

with that spacing and format. The first underline would be a character, and the second would be a number. I arrived at that conclusion because %c and %d are character and double flags respectively, so it actually writes: `x=(x-‘c’+‘C’), y=((double) (x-y))\n`

I don’t know what x-‘c’+’C’ ends up being, as it is character arithmetic; characters are stored as ASCII code in C so it takes the numerical ASCII value of f, subtract it by the value of c, and adds the value of C, then converts it back into a character because of the %c flag. I also don’t know what x-y is, but it ends up being a number in the string because of the %d flag. \n means newline so the next printf is on a separate line. The same logic applies to the second printf but I think it’s the answer to the character arithmetic should be c since c should be 2 after a. The second print f also has an extra space at the front.

Code:

```c
char x = 'f';
char y = 'a';
printf("x=%c, y=%d \n", x - 'c' + 'C', x - y);
printf(" y+2 = %c\n", y + 2);
```

Output:

```
x=F, y=5
 y+2 = c
```

Q2

It should print:

```
|3.141590|
|  3.1|
|3.142|
|       3.1|
|   3.14159|
```

With that spacing and format. This is expected as the first printf says to print a with at least 5 digits, so it will just print the number as is. %f itself adds an extra decimal place so that's why it has an extra 0 at the end. The second printf %5.1f means at least 5 digits, 1 digit after the decimal, 4 digits in front, including the decimal point as a digit. %5.4g prints 5 total digits with 4 digits precision *including* the decimal dot. %10.1f prints the float with 1 digit after the decimal point and with at least 10 character width total. %10.6g prints 10 total digits with 6 digits precision including the decimal. All the printfs also have an extra space at the very front, in front of the | bar.

Code:

```
float a = 3.14159;
printf(" |%f|\n", a);
printf(" |%5.1f|\n", a);
printf(" |%5.4g|\n", a);
printf(" |%10.1f|\n", a);
printf(" |%10.6g|\n", a);
```

Output:

```
|3.141590|
|  3.1|
|3.142|
|       3.1|
|   3.14159|
```

Q3

It should print:

21

3

4

4

3

a = 3, and all the print statements are simple variable output + new line, so its just 3 * 7 + 6 % 2 / 2 = 21. C takes in orders of operations, so it becomes 21 + 0 / 2 = 21 since 3*7 = 21 and 6mod2 = 0. 21 + 0 = 21.

The next line prints 3 since the ++ operation is done after the old a variable is called, so it does not change.

The next line prints 4 since it was the addition operation from the previous line

The next line prints 4 since this time its ++a instead of a++, negating the minus operation from the previous line.

The last line prints 3, since this times --a instead of a--, which means it subtracts before it can pull the variable.

Code:

```
int a = 3;
printf("%d\n", a * 7 + 6 % 2 / 2);
printf("%d\n", a++);
printf("%d\n", a--);
printf("%d\n", ++a);
printf("%d\n", --a);
```

Output:

```
21
3
4
4
3
```

Q4

The first if statement `if(a&&b)` passes as a and b are both nonzero, only 0 is false, so it prints

The second if statement `if(a||b) passes as a or b is true, so it prints.

Then the a is set to 0 and b is set to 10, equivalent to to setting a to false and b to true.

The third if does not print as a is false

The forth if prints as b is true

The last if does not print as b is true, so not b is false.

So the output would be

```
a&&b (5 & 20) -- True
a||b (5 || 20) -- True
a||b (0 || 10) -- True
```

with the spacing and format.

Code:

```c
int a = 5, b = 20;
if (a && b)
{
    printf(" a&&b (5 & 20) -- True \n");
}
if (a || b)
{
    printf(" a||b (5 || 20) -- True \n");
}
a = 0, b = 10;
if (a && b)
{
    printf(" a&&b (0 && 10) -- True \n");
}
if (a || b)
{
    printf(" a||b (0 || 10) -- True \n");
}
if (!b)
{
    printf(" !b (!10) -- True \n");
}
```

Output:

```
a&&b (5 & 20) -- True
a||b (5 || 20) -- True
a||b (0 || 10) -- True
```

Q5

The first if statement, If (a<b), is false, given that a = 2 and b = -1, so it just goes to the print statement. The prints statement prints c, which is 2. Or the program doesn't compile since its not a one line if statement. I don't really know how it works since I've never tried like this.

Code:

```
int a = 2, b = -1, c = 2;

if (a < b)
    if (c < a)
        c = 0;
    else
        c += 3;

printf("%d \n", c);
```

Output:

2

It seems like it does compile and work properly since the if else statement can be written in one line, which means the entire big if statement can be written in one line, so the brackets aren't necessary.

Q6

a is 12, so 12 mod 3 should be 0, which means it falls under case 0: b + 3. But its written as b + 3 and not b = b+3 or b+=3, meaning it does nothing, so b is still equal to 1 after the operation. Then the program goes to the print statement, which just prints b as a value, so the expected output is:

1

Code:

```
int a = 12, b = 1;
switch (a % 3)
{
case 0:
    b + 3;
    break;
case 1:
    b--;
    break;
case 2:
    b++;
    break;
default:
    printf("Default.\n");
}
printf("%d\n", b);
```

Output:

1

Q7

a is an integer array of length 5, but sizeof(a) returns the byte size of a, not its length in C. Since each int uses 4 bytes of storage, sizeof(a) should be 20, therefore the first printf statement should print 20.

The second printf statement should print 4, since ints use 4 bytes.

And the third printf statement should print 5, since the number at index 4 of a is 5.

Expected output:

20

4

5

Code:

```
int a[] = {1, 2, 3, 4, 5};
printf("%d\n", sizeof(a));
printf("%ld\n", sizeof(a[1]));
printf("%d", a[4]);
```

Output:

```
20
4
5
```

Q8

The first printf statement should print 6, and the sizeof() function still gives the byte size of the thing that's passed to it, sizeof(c) should give 6, as chars only take up 1 byte of storage.

The second printf statement should print b, as it %c is a char flag, and c[1] is the char at index 1 in c, which is b.

The third printf statement should print the array starting from f had c ended with '\0', as %s is a string flag and tries to read c as a string, but because it doesn't end with the null character, it can print anything after ftpo.

The print statement should start printing at 'f' instead of 'a' because C as a language is very memory based, so c + 2 means: find the memory address of the first index, and add 2 to the memory address, which is f.

Expected output:

6

b

`ftpo___`

where ___ can be anything.

Code:

```c
#include <stdio.h>
#include <string.h>
int main()
{
    char c[] = {'a', 'b', 'f', 't', 'p', 'o'};
    printf("%ld \n", sizeof(c));
    printf("%c \n", c[1]);
    printf("%s \n", c + 2);
    return 0;
}
```

Output:

```
6
b
ftpo♀ a
```

It seems like it printed more than one random letter after it, but honestly it could've been anything.