P1

Given that a = 4, b=7, c=9:

1)

`&` is the bitwise AND operator, which compares each position of the binary data of two variables and returns a decimal number represented by a binary number with both positions are 1. Table:

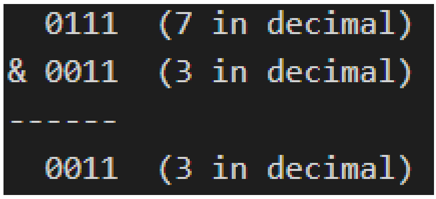
0 0 = 0

0 1 = 0

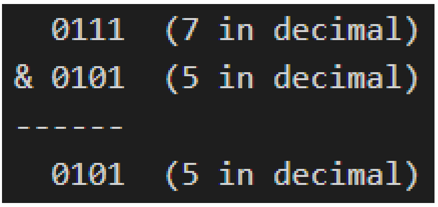
1 0 = 0

1 1 = 1

To visualize(written in notepad)



```
  0111   (7 in decimal)
& 0011   (3 in decimal)
------
  0011   (3 in decimal)
```

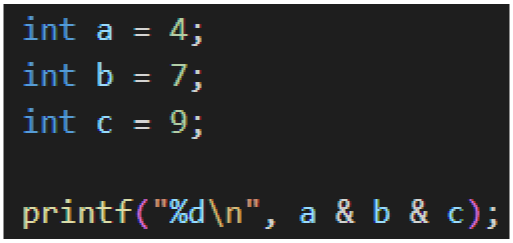


```
  0111   (7 in decimal)
& 0101   (5 in decimal)
------
  0101   (5 in decimal)
```

So a&b&c would output:

4&7&9 **-->** 0100 & 0111 & 1001 **-->** 0100 & 1001 **-->** 0

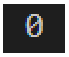
Code:



```
int a = 4;
int b = 7;
int c = 9;

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

Output:



```
0
```

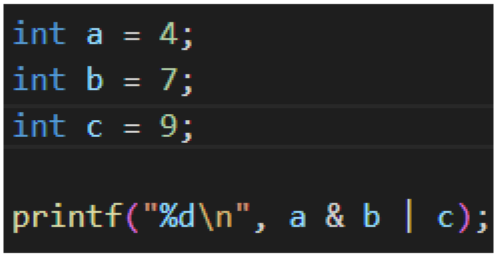
2)

`|` is the bitwise OR operator, so if either bit is 1, it is 1.

So the output would be:

4&7|9 **-->** 0100 & 0111 | 1001 **-->** 0100 | 1001 **-->** 1101 **-->** 13

Code:



```
int a = 4;
int b = 7;
int c = 9;

printf("%d\n", a & b | c);
```

Output:

```
13
```

3)

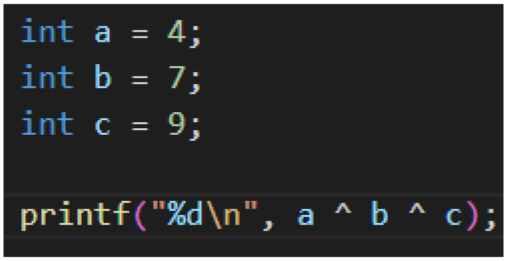
`^` is the bitwise XOR operator, it compares to see if the two bits are different. Table:

1 1 = 0
1 0 = 1
0 1 = 1
0 0 = 0

So the output would be:

4^7^9 **-->** 0100 ^ 0111 ^ 1001 **-->** 0011 ^ 1001 **-->** 1010 **-->** 10

Code:



```
int a = 4;
int b = 7;
int c = 9;

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

Output:

```
10
```

4)

`||` is the logical OR operator, where 0 is false and non zero is true. Table where x != 0:

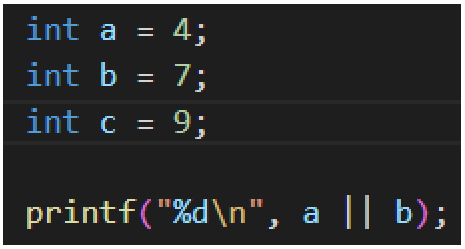
x x = 1
x 0 = 1
0 x = 1
0 0 = 0

So the output would be:

1

As a and b are both nonzero.

Code:



```
int a = 4;
int b = 7;
int c = 9;

printf("%d\n", a || b);
```

Output:

```
1
```

5)

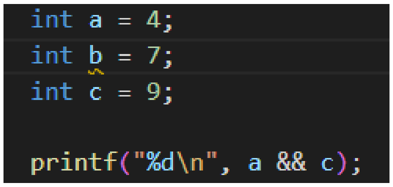
`&&` is the logical AND operator, where 0 is false and non zero is true. Table where x != 0:

x x = 1
x 0 = 0
0 x = 0
0 0 = 0

So the output would be:

1

Code:



```
int a = 4;
int b = 7;
int c = 9;

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

Output:

```
1
```

P2

1)

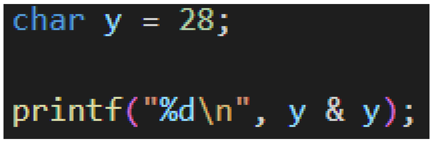
chars are stored as ASCII code in C, hence the initial value of char y = 28. Though I am not sure what char it is.

Like P1, & is the bitwise AND operator, which converts the decimal value of the char into binary, and compares it to itself. Since the operation is y & y, the output should be:

28

As the same binary number will have all the ones be at the same place.

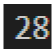
Code:



```
char y = 28;

printf("%d\n", y & y);
```

Output:



```
28
```

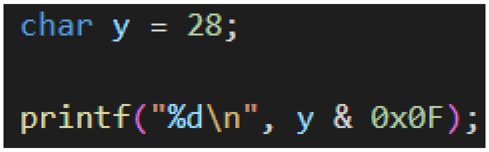
2)

0x0f is the hexadecimal value for the decimal number 15, as 0f is 15 and 0x is the prefix for hexadecimals in C.

So the output would be:

28 & 15 **-->** 11100 & 01111 **-->** 01100 **-->** 12

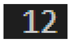
Code:



```
char y = 28;

printf("%d\n", y & 0x0F);
```

Output:



```
12
```

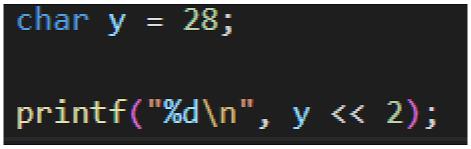
3)

The `<<` is a left shift operation, which shifts the bits of y two places to the left. Each shift to the left doubles the number, so y << 2 is equivalent to multiplying y by 4.

Which means the output is:

28 << 2 **-->** 11100 << 2 **-->** 1110000 **-->** 112

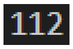
Code:



```
char y = 28;

printf("%d\n", y << 2);
```

Output:



```
112
```

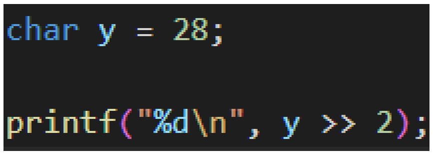
4)

The `>>` is a right shift operation, which shifts the bits of y two places to the right. Each shift to the right halves the number, so y >> 2 is equivalent to dividing y by 4.

Which means the output is:

28 >> 2 **-->** 11100 >> 2 **-->** 111 **-->** 7

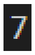
Code:



```
char y = 28;

printf("%d\n", y >> 2);
```

Output:



```
7
```

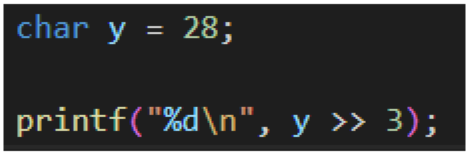
5)

Same as 4, but any shift that passes the “end point” is ignored.

Which means the output is:

28 >> 3 **-->** 11100 >> 3 **-->** 11~~100~~ **-->** 3

Code:



```
char y = 28;

printf("%d\n", y >> 3);
```

Output:

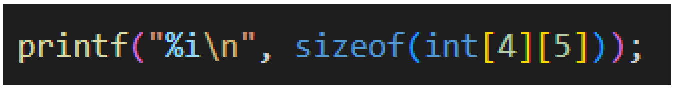
```
3
```

P3

1)

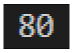
Each int is 4 bytes in size, and a 2D array with 4 rows and 5 columns would be able to store 20 things in it. The size of the array would be 20*4 = 80.

Code:



```
printf("%i\n", sizeof(int[4][5]));
```

Output:

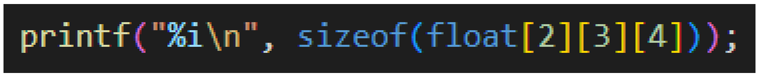


```
80
```

2)

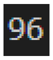
Each float is also 4 bytes in size, and a 3D array with dimensions 2,3,4 would be able to store 24 things. The size of the array would be 24 * 4 = 96.

Code:



```
printf("%i\n", sizeof(float[2][3][4]));
```

Output:

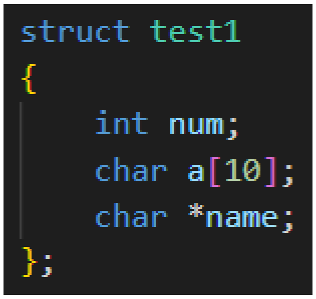


```
96
```

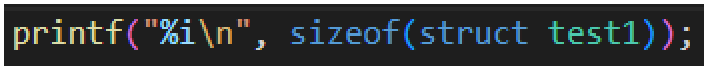
3)

Test1 contains an int, a char array with size 10, and a pointer. Chars are 1 byte each, so the array would be 10 bytes, the int would be 4 bytes, and the pointer should also be 4 bytes, for a total of 18 bytes. Each struct also has "padding" added for better memory access, but do I not know exactly how much padding is added to the struct, and I don't know how to calculate it.

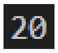
Code:



```
struct test1
{
    int num;
    char a[10];
    char *name;
};
```



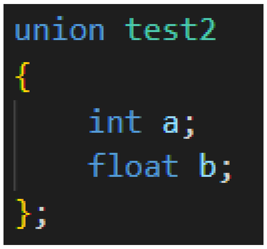
```
printf("%i\n", sizeof(struct test1));
```

Output:  20 It would seem that the padding was 2 bytes in this case.

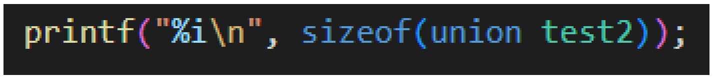
4)

Union in C shares its memeory between all variables of the union, meaning its size is determined by the size of the largest member. Since both int and float are 4 bytes, the size of test2 would be 4 bytes.

Code:



```
union test2
{
    int a;
    float b;
};
```



```
printf("%i\n", sizeof(union test2));
```

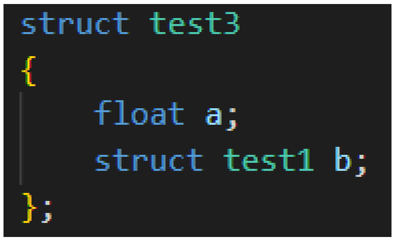
Output:

```
4
```

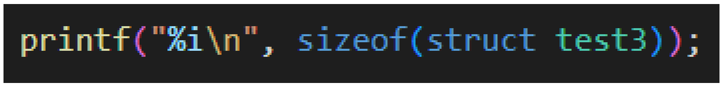
5)

Test3 contains a float and a test1 struct, since we know that the size of test1 is 20, we can say the sizeof test3 would be 20+4 = 24.

Code:

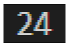


```
struct test3
{
    float a;
    struct test1 b;
};
```



```
printf("%i\n", sizeof(struct test3));
```

Output:



```
24
```