P1

Given that int i = 3, *p; p = &i;

p is a pointer variable. Since &i gives the address of i, p = &i would make p = the address of i.

Code implementation:

```
int i = 3, *p;
p = &i;
```

a) since the & symbol gives the address of the variable after it, the output would be the address of p. Since the address the computer decides to store the variables at is determined by the computer, I don't know what it is until I run the code.

Code:

```
printf("%p\n", (void *)&p);
```

I used %p since that is the pointer stand in, but since pointer values are ints, %i and %d or other number stand ins would work as well, but give a pure numerical value of the memory address.

Output:

```
0061FF18
```

b) Since i isn't a pointer variable, I'm pretty sure that there is no way to print *i and attempting to do it would not even compile.

Code:

```
printf("%i\n", *i);
```

And putting it in vscode, it does not compile and gives the error: error: invalid type argument of unary '*' (have 'int')

c) The * symbol, when used on a pointer, gives the value that is store at the address it points to. So if p is the pointer to i, then *p should be the value of the address of i.

This means that the output should be:

3

Since i=3.

Code:

```
printf("%i\n", *p);
```

I used %i because the variable type of what p is pointing to is an int.

Output:

```
3
```

d) Since p stores the address of i, trying to print p should just output the address of i, and has the same effect of printing &i. Again, since the address the computer decides to store the variables at is determined by the computer, I don't know what it is until I run the code.

Code:

```
printf("%p\n", (void *)p);
```

Same with a), I used %p since that is how you output pointer values, but %i or %d would still also work.

Output:

```
0061FF1C
```

e) i + 5 should be 3 + 5 which is 8, so the output should be 8.

Code:

```
printf("%i\n", i + 5);
```

Output:

```
8
```

P2

Given int a[5]={1,2,3,4,5},*p=a;

Pointer p = a sets p to the memory address of a. Since the array itself is stored as memory in c, we don't need to use p=&a and just p=a.

a) &a[1] gives the address of the second element in the array a.

Code:

```
printf("%p\n", (void*)&a[1]);
```

Output:

```
0061FF0C
```

b) p+2 is adding 2 to the address of a, this gives the address that is 2 after start of the array a, meaning output will be the same as &a[2].

Code:

```
printf("%p\n", (void*)(p + 2));

printf("%p\n", (void*)&a[2]);
```

Output:

```
0061FF10
0061FF10
```

I included the second printf to show that its the same as &a[2]

c) a++ should not compile because the array name a is not modifiable, so we can't increment the base address of an array. We could do p++, which has the same effect.

Code:

```
printf("%p\n", (void*)a++);
```

And it gives the error: error: lvalue required as increment operand when compiled

d) *a + 3 adds 3 to the value to the first element in the array a.

Since *a is the same as a[0], *a + 3 is 1 + 3 which is 4. So the expected output is 4.

Code:

```
printf("%i\n", *a + 3);
```

Output:

```
4
```

e) (*a)+3 is the same as d), since the * operation takes place before the + operation so adding the () doesn't not change anything.

Code:

```
printf("%i\n", (*a) + 3);
```

Output:

```
4
```

P3

a) Since ints are 4 bytes, p++ should shift the memory address by 4.

Code:

```
int a = 24, *p = &a;

printf("%d\n", (void*)p++);

printf("%d\n", (void*)p);
```

Output:

```
6422296
6422300
```

I used %d here since I thought it'd be easier to see that it moved by 4 bytes.

b) Since floats are also 4 bytes, it should just be the same as a)

Code:

```
float a = 24, *p = &a;

printf("%d\n", (void*)p++);

printf("%d\n", (void*)p);
```

Output:

```
6422296
6422300
```

c) Since doubles are 8 bytes, p++ should shift the memory address by 8.

Code:

```
double a = 24, *p = &a;

printf("%d\n", (void*)p++);

printf("%d\n", (void*)p);
```

Output:

```
6422288
6422296
```

d) Since cars are only 1 byte, p++ should just shift the memory address by 1.

Code:

```
char a = 24, *p = &a;

printf("%d\n", (void*)p++);

printf("%d\n", (void*)p);
```

Output:

```
6422299
6422300
```

e) int pointers are generally 4 bytes, so it's the same as a) again.

Code:

```
int x = 10;
int *a = &x;
int *p = &a;
printf("%d\n", (void *)p++);
printf("%d\n", (void *)p);
```

Output:

```
6422292
6422296
```

P4

Given int n=0,*p=&n,**q=&p;

a) *p gives the value that is store at the address it points to. So if p is the pointer to n, then *p should be the value of the address of n. Which means the output should be 0.

b) *n should not work for the same reasons as P1) b), as the * gives the value stored at the memory value of the variable after it, since n doesn't store a memory address, it won't work.

c) q should give the memory address of p (&p, which is what its set to in the init)

d) *q should give the value of p, which is the address of n. This should give the same thing as &n

e) **q gives the value of *p, as it goes *(*q) -> *(p) -> *(&n) -> n, which should give 0.