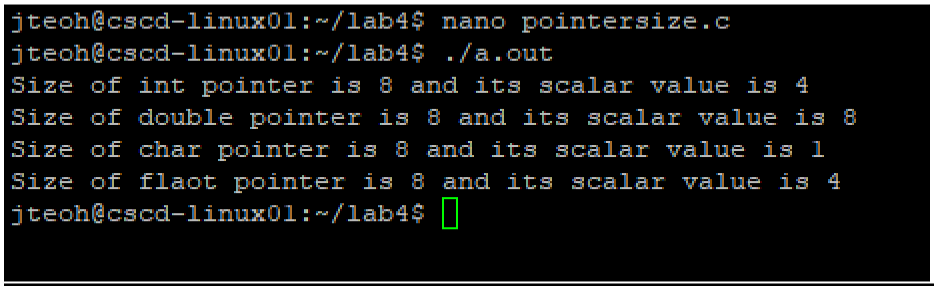
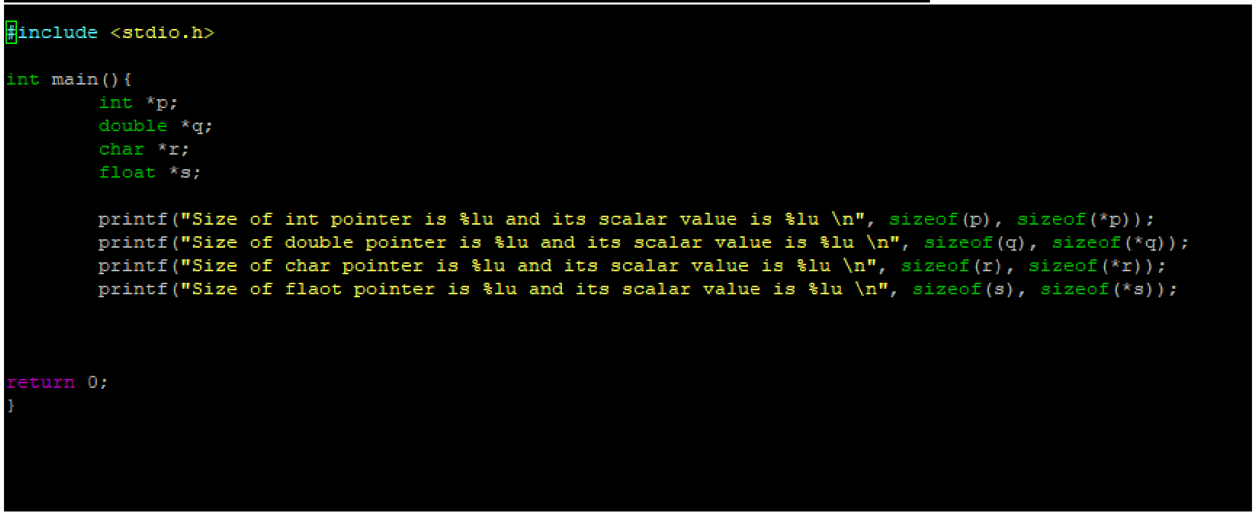
1) (A) Ptr = &c; this means ptr is pointing to address of variable c.
(B) ptr = &f; this tries to change ptr to point to the address of variable f, but variable f is of type int. this will not work.
(C) ptr = &'#'; this tries to change ptr to point to the address of a literal. This will not work.
(D) ptr = &500; this tries to change ptr to point to address of another literal, this will not work.
(E) ptr = &(f+3); this tries to change ptr to the resultant of the addition, which is a r value. This will not work.

2)



```
jteoh@cscd-linux01:~/lab4$ nano pointersize.c
jteoh@cscd-linux01:~/lab4$ ./a.out
Size of int pointer is 8 and its scalar value is 4
Size of double pointer is 8 and its scalar value is 8
Size of char pointer is 8 and its scalar value is 1
Size of flaot pointer is 8 and its scalar value is 4
jteoh@cscd-linux01:~/lab4$
```



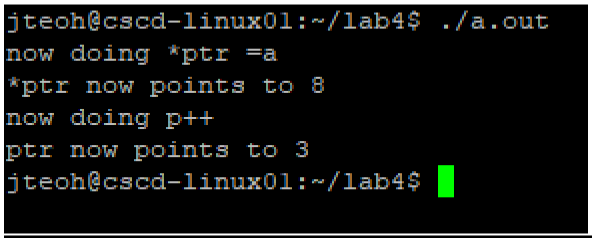
```c
#include <stdio.h>

int main(){
        int *p;
        double *q;
        char *r;
        float *s;

        printf("Size of int pointer is %lu and its scalar value is %lu \n", sizeof(p), sizeof(*p));
        printf("Size of double pointer is %lu and its scalar value is %lu \n", sizeof(q), sizeof(*q));
        printf("Size of char pointer is %lu and its scalar value is %lu \n", sizeof(r), sizeof(*r));
        printf("Size of flaot pointer is %lu and its scalar value is %lu \n", sizeof(s), sizeof(*s));



return 0;
}
```

The reason why all the size of various type pointers are the same is because the size of a pointer gives the size of the address the variable the pointer is pointing to. Thus they are all 8 bytes. This might change with other computers (64bit vs 32bit, etc). The scalar size is different as they refer to the size of the variable the pointer is pointing to. We can see that the int pointer has a scalar value of 4, which is the size of an int. The similar can be said about double, char and float, as the size of their respective pointers refer to the size of the variable they are pointing to.

3) The problem comes from the line
scanf("%d", &ptr);
as that tries to assign the address of pointer to whatever int you are passing in. To fix it, just remove & and it should work.

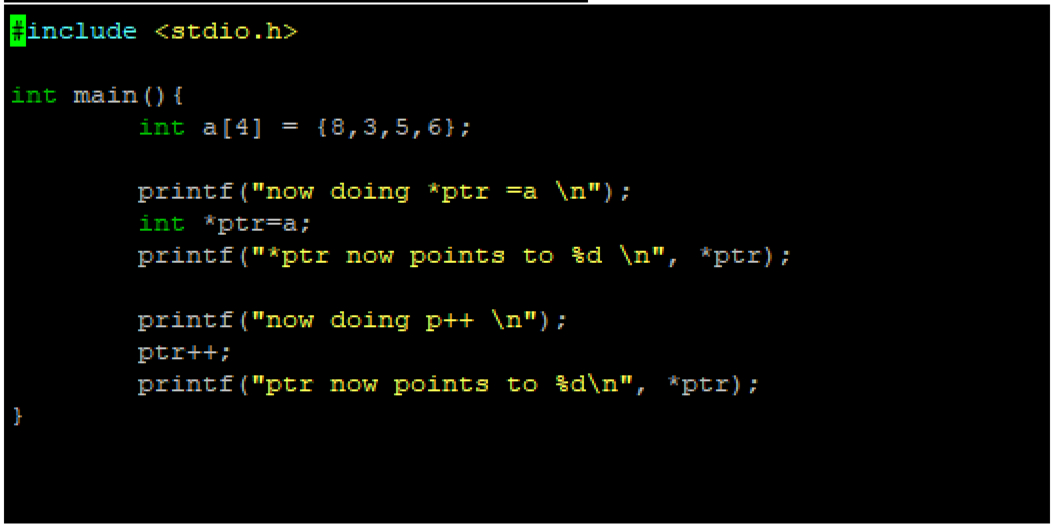
4) C = *p. c will equal to 10

***p = *p * *p.** a will equal 100.
(*p)++. a will still be 100, due to postfix, not prefix.
c = *&a. c will be 100.



```
jteoh@cscd-linux01:~/lab4$ ./a.out
now doing *ptr =a
*ptr now points to 8
now doing p++
ptr now points to 3
jteoh@cscd-linux01:~/lab4$
```

5)



```
#include <stdio.h>

int main(){
        int a[4] = {8,3,5,6};

        printf("now doing *ptr =a \n");
        int *ptr=a;
        printf("*ptr now points to %d \n", *ptr);

        printf("now doing p++ \n");
        ptr++;
        printf("ptr now points to %d\n", *ptr);
}
```

Int *ptr = a just assigns ptr to the address of the first element of array a, which is 8. Then when we do an increment, pointer moves the address up by the amount required to store an int, which moves the pointer up to the second element of the array, which is 3.

6) The first one is a non constant char array. The string hello world is copied to the array, which can then be modified via normal array methods. The second is a constant literal string. You are unable to modify that string at all.