After this statement owners will get all permissions, group will get read and execute permission, and others will get only read permission.

To grant execute permission to others we can write-

```
permission = permission | E_OTHERS;     /*Switch on 11th bit*/
```

To take away read permission from group we can write-

```
permission = permission & ~R_GROUP;     /*Switch off 4th bit*/
```

13.10 Bit Fields

We have seen how to access and manipulate individual bits or group of bits using bitwise operators. Bit fields provide an alternative method of accessing bits. A bit field is a collection of adjacent bits. It is defined inside a structure but is different from other members because its size is specified in terms of bits. The data type of a bit field can be int, signed int or unsigned int.

Let us take an example that shows the syntax of defining bit fields.

```
struct tag {
            unsigned a:2;
            unsigned b:5;
            unsigned c:1;
            unsigned d:3;
          };
```

Here the structure has four bit fields a, b, c and d. The sizes of a, b, c and d are 2, 5, 1 and 3 bits respectively. The bit fields can be accessed like other members of the structure using the dot operator. Whenever they appear inside expressions, they are treated like integers(of smaller range). These are some valid expressions using bit fields.

```
struct tag var;
var.a=2;
printf("%d",var.b);
x=var.a+var.b;
if(var.c==1)
    printf("Flag is on\n");
```

If the size of a bit field is n, then the range of values that the bit field can take is from 0 to 2^n

Bit field	Size in bits	Range of values
a	2	0 to 2^2-1(0 to 3)
b	5	0 to 2^5-1 (0 to 31)
c	1	0 to 2^1-1 (0 and 1)
d	3	0 to 2^3-1 (0 to 7)

It would be invalid to assign any value to a bitfield outside its range, for example-

```
var.b = 54;          /*Invalid*/
```

We can't apply sizeof and address operators to bit fields. So we can't use scanf to input the values in a bit field.

```
scanf("%d", &var.a);        /*Invalid*/
```

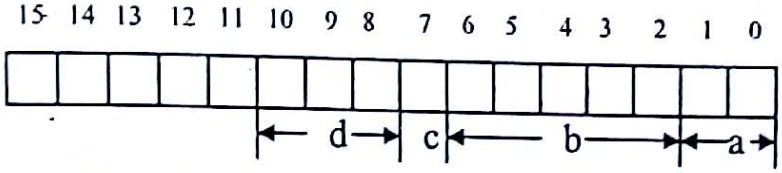
We may input the value into a temporary variable and then assign it to the bit field.

```
scanf("%d", &temp);
var.a = temp;
```

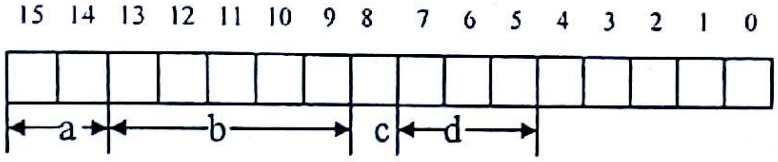
If we have pointer to structure then arrow operator(–>) can be used to access the bit fields. Code using bitfields is easier to understand than the equivalent masking operations, but bitfields are considered non-portable as most of the issues related with them are implementation dependent.

We had mentioned that the data type of bit fields can be int, signed int or unsigned int. A plain int field may be treated as signed by some compilers while as unsigned by others. So for portability, it is better to clearly specify signed or unsigned in the declaration of bit fields. If a bitfield is defined as signed, then it should be at least 2 bits long because one bit is used for sign.

The direction of allocation of bit fields within an integer is also implementation dependent. Some C compilers allocate the bit fields from right to left, while others may allocate them from left to right. If the bit fields are assigned from right to left, then the first field occupies the least significant bit. If the bit fields are assigned from left to right, then the first field occupies the most significant bit.



Four bit fields assigned from right to left

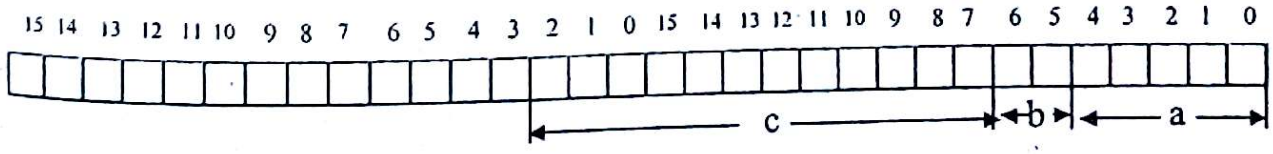


Four bit fields assigned from left to right

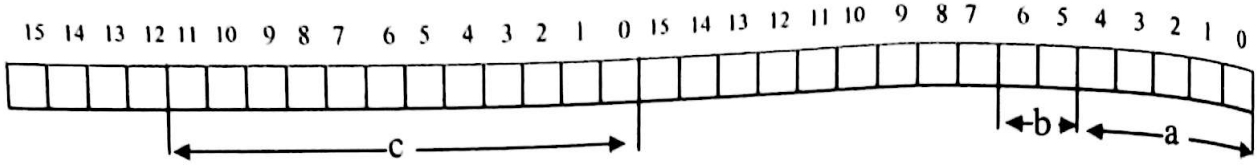
The other implementation dependent issue about bit fields is that whether they can cross integer boundaries or not. For example consider this structure-

```
struct tag{
            unsigned a:5;
            unsigned b:2;
            unsigned c:12;
        };
```

The first two fields occupy only 7 bits in a 16 bit integer, so 9 bits can still be used for another bit field. But the bits needed for next bit field is more than 9. Some implementations may start the next field(c) from a new integer, while others may just place the next field in 12 adjacent bits i.e. 9 unused from the previous integer and 3 bits from the next integer.



Bit fields cross integer boundary



Bit fields do not cross integer boundary

We can define unnamed bitfields for controlling the alignment of bitfields within an integer. The size of unnamed bitfield provides padding within the integer.

```
struct tag{
            unsigned a:5;
            unsigned b:2;
            unsigned  :9; /* padding within first integer */
            unsigned c: 12;
        };
```

Here the unnamed bitfield fills out the 9 unused bits of first integer, so the bitfield c starts with the second integer. Since these 9 bits don't have any name, so they can't be accessed from the program.

We can also take the size of the unnamed bitfield zero, so we have no need to provide the padding and next bitfield will start with second integer.

```
struct tag{
            unsigned a : 5;
            unsigned b : 2;
            unsigned   : 0;
            unsigned c : 12; /*this field starts from next integer*/
        };
```

Now we'll take the example of file permissions that we had seen earlier using bitwise operators.

```
struct permission
{
    unsigned r_owner:1;
    unsigned w_owner:1;
    unsigned a_owner:1;
    unsigned e_owner:1;
    unsigned r_group:1;
    unsigned w_group:1;
    unsigned a_group:1;
    unsigned e_group:1;
    unsigned r_others:1;
    unsigned w_others:1;
    unsigned a_others:1;
    unsigned e_others:1;
};
```

struct permissions perm = {1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0} ;

After this statement owners will get all permissions, group will get read and execute permission, and others will get only read permission.

Now we can write statements to grant and take permissions like this-

perm.e_others = 1; /*grant execute permission to others*/