Bit Fields in C

In C, we can specify the size (in bits) of the structure and union members. The idea of bit-field is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range. C Bit fields are used when the storage of our program is limited.

Need of Bit Fields in C

- *Reduces memory consumption.*
- *To make our program more efficient and flexible.*
- *Easy to Implement.*

Declaration of C Bit Fields

Bit-fields are variables that are defined using a predefined width or size. Format and the declaration of the bit-fields in C are shown below:

Syntax of C Bit Fields

```
struct
{
    data_type member_name : width_of_bit-field;
};
```

where,

- **data_type:** It is an integer type that determines the bit-field value which is to be interpreted. The type may be int, signed int, or unsigned int.
- **member_name:** The member name is the name of the bit field.
- **width_of_bit-field:** The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

Applications of C Bit Fields

- If storage is limited, we can go for bit-field.
- When devices transmit status or information encoded into multiple bits for this type of situation bit-field is most efficient.
- Encryption routines need to access the bits within a byte in that situation bit-field is quite useful.

Example of C Bit Fields

In this example, we compare the size difference between the structure that does not specify bit fields and the structure that has specified bit fields.

Structure Without Bit Fields

Consider the following declaration of date without the use of bit fields.

```c
// C Program to illustrate the structure without bit field

#include <stdio.h>



// A simple representation of the date

struct date {

    unsigned int d;

    unsigned int m;

    unsigned int y;

};



int main()

{

    // printing size of structure

    printf("Size of date is %lu bytes\n",

           sizeof(struct date));

    struct date dt = { 31, 12, 2014 };

    printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);

}
```

Output

```
Size of date is 12 bytes

Date is 31/12/2014
```

The above representation of 'date' takes 12 bytes on a compiler whereas an unsigned int takes 4 bytes. Since we know that the value of d is always from 1 to 31, and the value of m is from 1 to 12, we can optimize the space using bit fields.

Structure with Bit Field

The below code defines a structure named date with a single member month. The month member is declared as a bit field with 4 bits.

```
struct date
{
// month has value between 0 and 15,
// so 4 bits are sufficient for month variable.
    int month : 4;
};
```

However, if the same code is written using signed int and the value of the fields goes beyond the bits allocated to the variable, something interesting can happen.

Below is the same code but with signed integers:

- C

```
// C program to demonstrate use of Bit-fields

#include <stdio.h>



// Space optimized representation of the date

struct date {

    // d has value between 0 and 31, so 5 bits

    // are sufficient

    int d : 5;



    // m has value between 0 and 15, so 4 bits
    // are sufficient

    int m : 4;



    int y;

};



int main()

{

    printf("Size of date is %lu bytes\n",

           sizeof(struct date));

    struct date dt = { 31, 12, 2014 };

    printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);

    return 0;

}
```

Output

```
Size of date is 8 bytes

Date is -1/-4/2014
```

Explanation

The output comes out to be negative. What happened behind is that the value 31 was stored in 5 bit signed integer which is equal to 11111. The MSB is a 1, so it’s a negative number and you need to calculate the 2’s complement of the binary number to get its actual value which is what is done internally.

By calculating 2’s complement you will arrive at the value 00001 which is equivalent to the decimal number 1 and since it was a negative number you get a -1. A similar thing happens to 12 in which case you get a 4-bit representation as 1100 and on calculating 2’s complement you get the value of -4.

•

```
#include <stdio.h>


struct test {

    // Unsigned integer member x

    unsigned int x;

    // Bit-field member y with 33 bits

    unsigned int y : 33;

    // Unsigned integer member z

    unsigned int z;

};


int main()

{

    // Print the size of struct test

    printf("%lu", sizeof(struct test));


    return 0;

}
```