

## C - Bit Fields

Suppose your C program contains a number of TRUE/FALSE variables grouped in a structure called status, as follows –

```
struct {  
    unsigned int widthValidated;  
    unsigned int heightValidated;  
} status;
```

This structure requires 8 bytes of memory space but in actual, we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situations.

If you are using such variables inside a structure then you can define the width of a variable which tells the C compiler that you are going to use only those number of bytes. For example, the above structure can be re-written as follows –

```
struct {  
    unsigned int widthValidated : 1;  
    unsigned int heightValidated : 1;  
} status;
```

The above structure requires 4 bytes of memory space for status variable, but only 2 bits will be used to store the values.

If you will use up to 32 variables each one with a width of 1 bit, then also the status structure will use 4 bytes. However as soon as you have 33 variables, it will allocate the next slot of the memory and it will start using 8 bytes. Let us check the following example to understand the concept –

[Live Demo](#)

```
#include <stdio.h>  
#include <string.h>  
  
/* define simple structure */  
struct {
```

```
    unsigned int widthValidated;
    unsigned int heightValidated;
} status1;

/* define a structure with bit fields */
struct {
    unsigned int widthValidated : 1;
    unsigned int heightValidated : 1;
} status2;

int main( ) {
    printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
    printf( "Memory size occupied by status2 : %d\n", sizeof(status2));
    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

## Bit Field Declaration

The declaration of a bit-field has the following form inside a structure –

```
struct {
    type [member_name] : width ;
};
```

The following table describes the variable elements of a bit field –

Sr.No.	Element & Description
1	<b>type</b> An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int.
2	<b>member_name</b> The name of the bit-field.
3	<b>width</b> The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

The variables defined with a predefined width are called **bit fields**. A bit field can hold more than a single bit; for example, if you need a variable to store a value from 0 to 7, then you can define a bit field with a width of 3 bits as follows –

```
struct {  
    unsigned int age : 3;  
} Age;
```

The above structure definition instructs the C compiler that the age variable is going to use only 3 bits to store the value. If you try to use more than 3 bits, then it will not allow you to do so. Let us try the following example –

[Live Demo](#)

```
#include <stdio.h>  
#include <string.h>  
  
struct {  
    unsigned int age : 3;  
} Age;  
  
int main( ) {  
  
    Age.age = 4;  
    printf( "Sizeof( Age ) : %d\n", sizeof(Age) );  
    printf( "Age.age : %d\n", Age.age );  
}
```

```
Age.age = 7;
printf( "Age.age : %d\n", Age.age );

Age.age = 8;
printf( "Age.age : %d\n", Age.age );

return 0;
}
```

When the above code is compiled it will compile with a warning and when executed, it produces the following result –

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

---

---