International Journal of Engineering, Science and Mathematics
Vol. 10 Issue 06, June 2021,
ISSN: 2320-0294 Impact Factor: 6.765
Journal Homepage: http://www.ijmra.us, Email: editorijmie@gmail.com
Double-Blind Peer Reviewed Refereed Open Access International Journal - Included in the International Serial Directories Indexed & Listed at: Ulrich's Periodicals Directory ©, U.S.A., Open J-Gage as well as in Cabell's Directories of Publishing Opportunities, U.S.A

C –language- Bit Fields

AUTHOR-S.RANICHANDRA
CO-AUTHOR- DR. R. SUTHAGAR
DR S.DINESH

INTRODUCTION-

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

```
struct {
unsignedintwidthValidated;
unsignedintheightValidated;
} 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 {
unsignedintwidthValidated : 1;
unsignedintheightValidated : 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 */
```

International Journal of Engineering, Science and Mathematics
Vol. 10 Issue 06, June 2021,
ISSN: 2320-0294 Impact Factor: 6.765
Journal Homepage: http://www.ijmra.us, Email: editorijmie@gmail.com
Double-Blind Peer Reviewed Refereed Open Access International Journal - Included in the International Serial Directories Indexed & Listed at: Ulrich's Periodicals Directory ©, U.S.A., Open J-Gage as well as in Cabell's Directories of Publishing Opportunities, U.S.A

```
struct {
unsignedintwidthValidated;
unsignedintheightValidated;
} status1;

/* define a structure with bit fields */
struct {
unsignedintwidthValidated : 1;
unsignedintheightValidated : 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	**type** An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int.
2	**member_name** The name of the bit-field.

3	**width** 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 {
unsignedint 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 {
unsignedint 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
```

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_typemember_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

structdate {

   unsigned intd;

   unsigned intm;

   unsigned inty;
};



intmain()

{

   // printing size of structure

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

        sizeof(structdate));

   structdate 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

structdate {

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

   // are sufficient

   intd : 5;



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

   // are sufficient

   intm : 4;
```

```
    inty;

};



intmain()

{

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

           sizeof(structdate));

    structdate dt = { 31, 12, 2014 };

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

    return0;

}
```

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.

Interesting Facts About C Bit Fields

1. A special unnamed bit field of size 0 is used to force alignment on the next boundary.

Example: The below code demonstrates how to force alignment to the next memory boundary using bit fields.

- C

```
// C Program to illustrate the use of forced alignment in

// structure using bit fields

#include <stdio.h>



// A structure without forced alignment

structtest1 {

    unsigned intx : 5;

    unsigned inty : 8;

};



// A structure with forced alignment

structtest2 {
    unsigned intx : 5;

    unsigned int: 0;

    unsigned inty : 8;

};



intmain()

{

    printf("Size of test1 is %lu bytes\n",

           sizeof(structtest1));

    printf("Size of test2 is %lu bytes\n",

           sizeof(structtest2));

    return0;

}
```

Output

```
Size of test1 is 4 bytes

Size of test2 is 8 bytes
```

2. We cannot have pointers to bit field members as they may not start at a byte boundary.

Example: The below code demonstrates that taking the address of a bit field member directly is not allowed.

- C

```
// C program to demonstrate that the pointers cannot point

// to bit field members

#include <stdio.h>

structtest {

    unsigned intx : 5;

    unsigned inty : 5;

    unsigned intz;

};

intmain()

{

    structtest t;



    // Uncommenting the following line will make

    // the program compile and run

    printf("Address of t.x is %p", &t.x);
    // The below line works fine as z is not a

    // bit field member

    printf("Address of t.z is %p", &t.z);

    return0;

}
```

Output

```
prog.c: In function 'main':

prog.c:14:1: error: cannot take address of bit-field 'x'

printf("Address of t.x is %p", &t.x);
```

3. It is implementation-defined to assign an out-of-range value to a bit field member.

Example: The below code demonstrates the usage of bit fields within a structure and assigns an out-of-range value to one of the bit field members.

- C

```
// C Program to show what happends when out of range value

// is assigned to bit field member

#include <stdio.h>
```

```
structtest {

    // Bit-field member x with 2 bits

    unsigned intx : 2;

    // Bit-field member y with 2 bits

    unsigned inty : 2;

    // Bit-field member z with 2 bits

    unsigned intz : 2;

};



intmain()

{

    // Declare a variable t of type struct test

    structtest t;

    // Assign the value 5 to x (2 bits)

    t.x = 5;



    // Print the value of x
    printf("%d", t.x);



    return0;

}
```

Output
Implementation-Dependent

4. Array of bit fields is not allowed.

Example: The below C program defines an array of bit fields and fails in the compilation.

- C

```

// C Program to illustrate that we cannot have array bit

// field members

#include <stdio.h>



// structure with array bit field

structtest {

    unsigned intx[10] : 5;

};
```

```
intmain() {}
```

Output

```
prog.c:3:1: error: bit-field 'x' has invalid type

unsignedint x[10]: 5;

 ^
```

Assume that unsigned int takes 4 bytes and long int takes 8 bytes.
Ans:

- *C*

```
#include <stdio.h>



structtest {

    // Unsigned integer member x

    unsigned intx;

    // Bit-field member y with 33 bits

    unsigned inty : 33;

    // Unsigned integer member z

    unsigned intz;

};
```

```
intmain()

{

    // Print the size of struct test

    printf("%lu", sizeof(structtest));



    return0;

}
```

Error:
./3ec6d9b7-7ae2-411a-a60e-66992a7fc29b.c:4:5: error: width of 'y' exceeds its type

unsignedint y : 33;

```
    ^
```

Predict the output of the following program.

Ans:

- *C*

```
#include <stdio.h>



structtest {
    // Unsigned integer member x

    unsigned intx;

    // Bit-field member y with 33 bits

    longinty : 33;

    // Unsigned integer member z

    unsigned intz;

};



intmain()

{

    // Declare a variable t of type struct test

    structtest t;

    // Pointer to unsigned int, pointing to member x

    unsigned int* ptr1 = &t.x;

    // Pointer to unsigned int, pointing to member z

    unsigned int* ptr2 = &t.z;
    // Print the difference between the two pointers

    printf("%d", ptr2 - ptr1);



    return0;

}
```

Output
4

Q3. Predict the output of the following program.

Ans:

- *C*

```
#include <stdio.h>



uniontest {

    // Bit-field member x with 3 bits

    unsigned intx : 3;

    // Bit-field member y with 3 bits

    unsigned inty : 3;
    // Regular integer member z

    intz;

};



intmain()

{

    // Declare a variable t of type union test

    uniontest t;

    // Assign the value 5 to x (3 bits)

    t.x = 5;

    // Assign the value 4 to y (3 bits)

    t.y = 4;

    // Assign the value 1 to z (32 bits)

    t.z = 1;

    // Print the values of x, y, and z

    printf("t.x = %d, t.y = %d, t.z = %d", t.x, t.y, t.z);
    return0;

}
```

Output

```
t.x = 1, t.y = 1, t.z = 1
```

REFERANCES-

1.GNU-Reference manual.

2.Wikipedia-BIT FIELDS.

3.BIT FIELDS IN C-JAVAT POINT.

4.CPP-Reference.

5.Greeks Tutorial material search.