



**Department of Computer Science and Engineering,
PES University, Bangalore, India**

**Lecture Notes
Problem Solving With C
UE24CS151B**

***Lecture #16
Bit Fields in C***

**By,
Prof. Sindhu R Pai,
Theory Anchor, Feb-May, 2025
Assistant Professor
Dept. of CSE, PESU**

**Many Thanks to
Dr. Shylaja S S (Director, CCBD and CDSAML Research Center, PES University)
Prof. Nitin V Pujari (Dean, Internal Quality Assurance Cell, PES University)**

Unit #: 3**Unit Name: Text Processing and User-Defined Types****Topic: Bit fields in C**

Course objectives: The objective(s) of this course is to make students

- Acquire knowledge on how to solve relevant and logical problems using computing Machine.
- Map algorithmic solutions to relevant features of C programming language constructs.
- Gain knowledge about C constructs and its associated ecosystem.
- Appreciate and gain knowledge about the issues with C Standards and it's respective behaviours.

Course outcomes: At the end of the course, the student will be able to:

- Understand and Apply algorithmic solutions to counting problems using appropriate C Constructs.
- Understand, Analyze and Apply sorting and Searching techniques.
- Understand, Analyze and Apply text processing and string manipulation methods using Arrays, Pointers and functions.
- Understand user defined type creation and implement the same using C structures, unions and other ways by reading and storing the data in secondary systems which are portable.

Sindhu R Pai

Theory Anchor, Feb - May, 2025

Dept. of CSE,

PES University

Introduction

In C programming, Bit field is a data structure that **allows the programmer to allocate memory to structures and unions in bits** in order to utilize computer memory in an efficient manner. This is especially useful when memory conservation is critical, such as in **embedded systems or when dealing with hardware registers**. Instead of allocating a full int (typically 4 bytes) to every variable, **bit fields let you store data with as few bits as needed**. In programming terminology, a bit field is of great significance due to following reasons:

- You can efficiently pack multiple variables into a single word.
- Controlling the memory layout at the bit level is possible.
- You can represent flags or values that need only a few bits (e.g., 0–7 using 3 bits).

The idea 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. **The variables defined with a predefined width are called bit fields**. A bit field can hold more than a single bit.

Syntax: `struct structure_name {
 data_type member_name : number_of_bits;
 };`

data_type: Typically int, unsigned int, or signed int. Using char or short may lead to implementation-defined behavior.

member_name: Name of the bit-field.

number_of_bits: The exact number of bits to allocate for that member (a positive integer).

Example: `struct Flags {
 unsigned int isVisible : 1;
 unsigned int isEnabled : 1;
 unsigned int errorCode : 3;
 };`

- isVisible and isEnabled use 1 bit each (ideal for true/false flags).
- errorCode uses 3 bits (can represent values from 0 to 7).
- This structure will use only 5 bits (plus padding as needed by the compiler). Max 1 byte, instead of 12 or more bytes if defined without bit fields.

Characteristics/Properties of Bit Fields:

- **Used Within Structures or Unions Only:** Bit fields can only be defined as members of a struct. You cannot declare a bit field outside.
- **Specify Memory in Bits:** Bit fields allow you to allocate a specific number of bits (not bytes) to a structure member.
- **Typically int, signed int, or unsigned int:** Although other integer types can be used, int, signed int, and unsigned int are most common and portable. Usage of char or short is compiler-dependent.
- **Maximum Width = Size of Base Type:** The number of bits assigned to a bit field must not exceed the size (in bits) of the underlying data type
- **Automatically Packed (with Padding Possible):** The compiler packs bit fields into as few memory units as possible, but may introduce padding to maintain alignment. The order in which bit fields are packed (left-to-right or right-to-left) is compiler-specific and can affect portability.
- **Unnamed Bit Fields for Padding:** Anonymous bit fields (without a name) can be used to add padding or align other fields:
- **No Address Access:** You cannot take the address of a bit-field member using &, since it may not have a distinct memory address.
- **Cannot Be Arrays or Pointers:** Bit field members cannot be declared as arrays or pointers.
- **Storage Class Restrictions:** Bit fields cannot be declared with the static or extern storage class within a structure definition. These qualifiers apply to the structure variable, not to individual bit-field members.

Coding Example_1: All these examples are run on Windows 32 bit machine

```
#include<stdio.h>

struct Status
{
    unsigned int bin1:1; // 1 bit is allocated for bin1. only two digits can be stored 0 &1.
}s};
```

```
int main()
{
    printf("Size of structure is %lu\n",sizeof(struct Status));
    struct Status s;
    //scanf("%d",&s.bin1); // Error
    s.bin1 = 0;    printf("%d\n",s.bin1);
    s.bin1 = 1;    printf("%d",s.bin1);
    return 0;
}
```

```
C:\Users\Dell>a
Size of structure is 4
0
1
```

The maximum value that can be stored in bit fields depends on whether the bit field has enough memory to hold the value. If it can't fit, results in **compile time warning**. **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.**

Coding Example_2: The age variable is going to use only 3 bits to store the value. If you try to store the value with more than 3 bits, it doesn't allow.

```
#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;
}
```

```
C:\Users\Dell>gcc -c check.c
check.c: In function 'main':
check.c:12:14: warning: large integer implicitly truncated to unsigned type [-Woverflow]
    Age.age = 8;
             ^
```

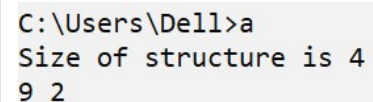
```
C:\Users\Dell>a
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

Coding Example_3: Assigning signed value to an unsigned variable for which the bit fields are specified.

```
#include<stdio.h>

struct Status
{
    unsigned int bin1:4; // 4 bits is allocated for bin1. 0 to 15, any number can be used.
    unsigned int bin2:2; // 2 bits allocated for bin2. 0 to 3, any number can be used
};

int main()
{
    printf("Size of structure is %lu\n",sizeof(struct Status)); // again 4 bytes
    struct Status s1;
    s1.bin1=-7; // it is unsigned. but sign is given while assigning
    s1.bin2=2;
    printf("%d %d",s1.bin1,s1.bin2); //
    return 0;
}
```



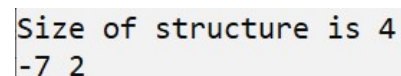
```
C:\Users\Dell>a
Size of structure is 4
9 2
-
```

Coding Example_4: Structure having two members with signed and unsigned variable for which bit fields are specified.

```
#include<stdio.h>

struct Status
{
    int bin1:4; // one bit is used for representing the sign. Another 3 bits for data
    unsigned int bin2:2; // 2 bits allocated for bin2. 0 to 3, any number can be used
};

int main()
{
    printf("Size of structure is %lu\n",sizeof(struct Status)); // again 4 bytes
    struct Status s1; s1.bin1=-7; s1.bin2=2;
    printf("%d %d",s1.bin1,s1.bin2);
    return 0;
}
```



```
Size of structure is 4
-7 2
```

Unnamed Bit Fields

Bit-field **members declared without an identifier** (i.e., no variable name) are unnamed bit fields. They are primarily used for:

Padding: To align the next member on a specific boundary.

Skipping bits: When certain bits are reserved or not used.

```
struct example {  
    unsigned int a : 3;  
    unsigned int : 2; // unnamed 2-bit field (padding)  
    unsigned int b : 4;  
}; // a uses 3 bits, 2 bits are skipped (no name, so not accessible) and b uses 4 bits.
```

Bit fields with a length of 0 must be unnamed. Unnamed bit fields cannot be referenced or initialized. A zero-width bit field can cause the next field to be aligned on the next container boundary where the container is the same size as the underlying type of the bit field.

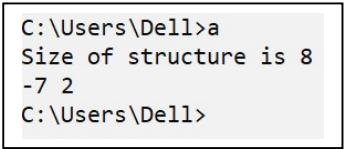
Points to note:

- You **cannot access unnamed bit fields** — they don't have a name.
- Their only purpose is **alignment or spacing** within the memory layout.
- They help in **matching memory layouts** with hardware registers or protocols.
- The bit width **must be specified** even for unnamed fields.

Coding Example_5:

```
struct Status  
{  
    int bin1:4;    // one bit is used for representing the sign. Another 3 bits for dat  
    //int i:0;// you cannot allocate 0 bits for any member inside the structure. Error  
    int :0;// A special unnamed bit field of size 0 is used to force alignment on  
    // next boundary. observe no member variable. only size is 0 bit  
    unsigned int bin2:2; // 2 bits allocated for bin2. 0 to 3, any number can be used  
};
```

```
int main()
{
    // observe the size of the structure. Now 8 bytes
    printf("Size of structure is %lu\n", sizeof(struct Status));
    struct Status s1;
    s1.bin1=-7; s1.bin2=2;
    printf("%d %d", s1.bin1, s1.bin2);
    return 0;
}
```



```
C:\Users\Dell>a
Size of structure is 8
-7 2
C:\Users\Dell>
```

Invalid Usage of Bit Fields in C

Coding Example_6:

```
#include <stdio.h>
struct InvalidBitFields {
    int a[5] : 3;        // ✗Error: Bit field cannot be an array
    int *ptr : 4;         // ✗Error: Bit field cannot be a pointer
    static int b : 2;     // ✗Error: Bit field cannot have static storage class
    unsigned int data : 40; // ✗Error: Width exceeds the size of base type (usually 32 bits)
};
int main() {
    struct InvalidBitFields ib;
    ib.data = 100;
    printf("Invalid Bit Fields: %u\n", ib.data);
    return 0;
}
```

Bit fields offer a **smart way to manage memory efficiently by packing data tightly**, especially useful in **systems with limited resources**. Whether it's setting flags or designing hardware-friendly structures, bit fields help you write compact, efficient C code.

Use bit fields — save space, code efficiently!!