Typedef:

The **typedef** is a keyword that is used to provide existing data types with a new name. The C typedef keyword is used to redefine the name of already existing data types. When names of datatypes become difficult to use in programs, typedef is used with user-defined datatypes, which behave similarly to defining an alias for commands.

Declaration of Bit Fields in a typedef:

To declare bit fields inside a typedef, you define a struct with bit-field members and then use typedef to create an alias for it.

```c
#include <stdio.h>

typedef struct {
    unsigned int flag1 : 1;   // 1-bit field
    unsigned int flag2 : 3;   // 3-bit field
    unsigned int flag3 : 4;   // 4-bit field
} BitFieldStruct;

int main() {
    BitFieldStruct b;
    b.flag1 = 1;
    b.flag2 = 5;  // 5 fits in 3 bits (101)
    b.flag3 = 12;  // 12 fits in 4 bits (1100)

    printf("Size of BitFieldStruct: %zu bytes\n", sizeof(BitFieldStruct));
    return 0;
}
```

What Happens When a Bit Field Overflows:

If you assign a value to a bit field that exceeds its declared width, the behavior depends on whether the bit field is **signed** or **unsigned**.

1. Unsigned Bit Fields (Most Common Case)

- The value is **truncated** to fit within the specified number of bits.
- Only the **least significant bits (LSB)** are kept.

2. Signed Bit Fields

- If the bit field is **signed**, the behavior depends on **sign representation** (usually **two's complement**).
- The **most significant bit (MSB)** determines the sign.
- If the value overflows, it **wraps around** in a signed manner.
-

Typedef is used to create an alias for complex types (like struct, union, or function pointers) to simplify their usage.

typedef **with** struct:

```c
// Without typedef (must use 'struct' keyword every time)
struct Point {
    int x, y;
};
struct Point p1;  // Declaration requires 'struct'

// With typedef (simpler usage)
typedef struct {
    int x, y;
} Point;
Point p2;  // No need for 'struct' keyword
```

typedef **with** union:

```c
typedef union {
    int i;
    float f;
    char c;
} Data;
Data d;
d.i = 42;
```

typedef **with function pointers:**

```c
typedef int (*MathFunc)(int, int);  // Defines a function pointer type
int add(int a, int b) { return a + b; }
MathFunc operation = add;  // Now 'operation' can be used like a function
```

What is the default underlying type of an enum**?**

In C, the **underlying type of an** enum **is** int **by default**, but the actual storage size may vary depending on the compiler and the range of values.

- Each enum constant is an int (usually 4 bytes).
- Some compilers optimize storage if the values fit in a smaller type (char for 0-255).

When is using a union **more memory-efficient?**

A union is useful when:

1. **Only one member is needed at a time** (a variable that can be int, float, or char).
2. **Memory optimization is critical** (embedded systems, network protocols).
3. **Implementing variant types** (parsing different data formats).