Chapter 9: Bit-fields

Parameter	Description
type-specifier	`signed`, `unsigned`, `int` or `_Bool`
identifier	The name for this field in the structure
size	The number of bits to use for this field

Most variables in C have a size that is an integral number of bytes. Bit-fields are a part of a structure that don't necessarily occupy a integral number of bytes; they can any number of bits. Multiple bit-fields can be packed into a single storage unit. They are a part of standard C, but there are many aspects that are implementation defined. They are one of the least portable parts of C.

Section 9.1: Bit-fields

A simple bit-field can be used to describe things that may have a specific number of bits involved.

```
struct encoderPosition {
   unsigned int encoderCounts : 23;
   unsigned int encoderTurns  : 4;
   unsigned int _reserved     : 5;
};
```

In this example we consider an encoder with 23 bits of single precision and 4 bits to describe multi-turn. Bit-fields are often used when interfacing with hardware that outputs data associated with specific number of bits. Another example could be communication with an FPGA, where the FPGA writes data into your memory in 32 bit sections allowing for hardware reads:

```
struct FPGAInfo {
    union {
        struct bits {
            unsigned int bulb1On  : 1;
            unsigned int bulb2On  : 1;
            unsigned int bulb1Off : 1;
            unsigned int bulb2Off : 1;
            unsigned int jetOn    : 1;
        };
        unsigned int data;
   };
};
```

For this example we have shown a commonly used construct to be able to access the data in its individual bits, or to write the data packet as a whole (emulating what the FPGA might do). We could then access the bits like this:

```
FPGAInfo fInfo;
fInfo.data = 0xFF34F;
if (fInfo.bits.bulb1On) {
    printf("Bulb 1 is on\n");
}
```

This is valid, but as per the C99 standard 6.7.2.1, item 10:

> The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined.

You need to be aware of endianness when defining bit-fields in this way. As such it may be necessary to use a preprocessor directive to check for the endianness of the machine. An example of this follows:

```
typedef union {
    struct bits {
#if defined(WIN32) || defined(LITTLE_ENDIAN)
    uint8_t commFailure :1;
    uint8_t hardwareFailure :1;
    uint8_t _reserved :6;
#else
    uint8_t _reserved :6;
    uint8_t hardwareFailure :1;
    uint8_t commFailure :1;
#endif
    };
    uint8_t data;
} hardwareStatus;
```

Section 9.2: Using bit-fields as small integers

```
#include <stdio.h>

int main(void)
{
    /* define a small bit-field that can hold values from 0 .. 7 */
    struct
    {
        unsigned int uint3: 3;
    } small;

    /* extract the right 3 bits from a value */
    unsigned int value = 255 - 2; /* Binary 11111101 */
    small.uint3 = value;          /* Binary      101 */
    printf("%d", small.uint3);

    /* This is in effect an infinite loop */
    for (small.uint3 = 0; small.uint3 < 8; small.uint3++)
    {
        printf("%d\n", small.uint3);
    }

    return 0;
}
```

Section 9.3: Bit-field alignment

Bit-fields give an ability to declare structure fields that are smaller than the character width. Bit-fields are implemented with byte-level or word-level mask. The following example results in a structure of 8 bytes.

```
struct C
{
    short s;            /* 2 bytes */
    char  c;            /* 1 byte */
    int   bit1 : 1;     /* 1 bit */
    int   nib  : 4;     /* 4 bits padded up to boundary of 8 bits. Thus 3 bits are padded */
    int   sept : 7;     /* 7 Bits septet, padded up to boundary of 32 bits. */
};
```

The comments describe one possible layout, but because the standard says *the alignment of the addressable storage unit is unspecified*, other layouts are also possible.

An unnamed bit-field may be of any size, but they can't be initialized or referenced.

A zero-width bit-field cannot be given a name and aligns the next field to the boundary defined by the datatype of the bit-field. This is achieved by padding bits between the bit-fields.

The size of structure 'A' is 1 byte.

```
struct A
{
    unsigned char c1 : 3;
    unsigned char c2 : 4;
    unsigned char c3 : 1;
};
```

In structure B, the first unnamed bit-field skips 2 bits; the zero width bit-field after `c2` causes `c3` to start from the char boundary (so 3 bits are skipped between `c2` and `c3`. There are 3 padding bits after `c4`. Thus the size of the structure is 2 bytes.

```
struct B
{
    unsigned char c1 : 1;
    unsigned char    : 2;    /* Skips 2 bits in the layout */
    unsigned char c2 : 2;
    unsigned char    : 0;    /* Causes padding up to next container boundary */
    unsigned char c3 : 4;
    unsigned char c4 : 1;
};
```

Section 9.4: Don'ts for bit-fields

1. Arrays of bit-fields, pointers to bit-fields and functions returning bit-fields are not allowed.
2. The address operator (&) cannot be applied to bit-field members.
3. The data type of a bit-field must be wide enough to contain the size of the field.
4. The `sizeof()` operator cannot be applied to a bit-field.
5. There is no way to create a `typedef` for a bit-field in isolation (though you can certainly create a `typedef` for a structure containing bit-fields).

```
typedef struct mybitfield
{
    unsigned char c1 : 20;   /* incorrect, see point 3 */
    unsigned char c2 : 4;    /* correct */
    unsigned char c3 : 1;
    unsigned int x[10]: 5;   /* incorrect, see point 1 */
} A;

int SomeFunction(void)
{
    // Somewhere in the code
    A a = { ... };
    printf("Address of a.c2 is %p\n", &a.c2);      /* incorrect, see point 2 */
    printf("Size of a.c2 is %zu\n", sizeof(a.c2)); /* incorrect, see point 4 */
}
```

Section 9.5: When are bit-fields useful?

A bit-field is used to club together many variables into one object, similar to a structure. This allows for reduced memory usage and is especially useful in an embedded environment.

```
e.g. consider the following variables having the ranges as given below.
a --> range 0 - 3
b --> range 0 - 1
c --> range 0 - 7
d --> range 0 - 1
e --> range 0 - 1
```

If we declare these variables separately, then each has to be at least an 8-bit integer and the total space required will be 5 bytes. Moreover the variables will not use the entire range of an 8 bit unsigned integer (0-255). Here we can use bit-fields.

```
typedef struct {
   unsigned int a:2;
   unsigned int b:1;
   unsigned int c:3;
   unsigned int d:1;
   unsigned int e:1;
} bit_a;
```

The bit-fields in the structure are accessed the same as any other structure. The programmer needs to take care that the variables are written in range. If out of range the behaviour is undefined.

```
int main(void)
{
   bit_a bita_var;
   bita_var.a = 2;              // to write into element a
   printf ("%d",bita_var.a);    // to read from element a.
   return 0;
}
```

Often the programmer wants to zero the set of bit-fields. This can be done element by element, but there is second method. Simply create a union of the structure above with an unsigned type that is greater than, or equal to, the size of the structure. Then the entire set of bit-fields may be zeroed by zeroing this unsigned integer.

```
typedef union {
    struct {
       unsigned int a:2;
       unsigned int b:1;
       unsigned int c:3;
       unsigned int d:1;
       unsigned int e:1;
    };
    uint8_t data;
} union_bit;
```

Usage is as follows

```
int main(void)
{
   union_bit un_bit;
   un_bit.data = 0x00;        // clear the whole bit-field
   un_bit.a = 2;              // write into element a
```

```
    printf ("%d",un_bit.a);     // read from element a.
    return 0;
}
```

In conclusion, bit-fields are commonly used in memory constrained situations where you have a lot of variables which can take on limited ranges.