


Declaring and Using Bit Fields


A structure or a C++ class can contain *bit fields* that allow you to access individual bits. You can use bit fields for data that requires just a few bits of storage. A bit field declaration contains a type specifier followed by an optional declarator, a colon, a constant expression, and a semicolon. The constant expression specifies how many bits the field reserves. A bit field that is declared as having a length of 0 causes the next field to be aligned on the next integer boundary. For a `_Packed` structure, a bit field of length 0 causes the next field to be aligned on the next byte boundary. Bit fields with a length of 0 must be unnamed. Unnamed bit fields cannot be referenced or initialized.

The maximum bit field length is implementation dependent. The maximum bit field length for the IBM C and C++ Compilers is 32 bits (4 bytes, or 1 word).

For portability, do not use bit fields greater than 32 bits in size.

The following restrictions apply to bit fields. You cannot:

- Define an array of bit fields
- Take the address of a bit field
- Have a pointer to a bitfield
-  Have a reference to a bit field

 You can declare a bit field as type `int`, signed `int`, or unsigned `int`. Bit fields of the type `int` are equivalent to those of type unsigned `int`.



Unlike ISO/ANSI C, C++ bit fields can be any integral type or enumeration type. When you assign a value that is out of range to a bit field, the low-order bit pattern is preserved and the appropriate bits are assigned.

If a series of bit fields does not add up to the size of an `int`, padding can take place. The amount of padding is determined by the alignment characteristics of the members of the structure. In some instances, bit fields can cross word boundaries.

The following example declares the identifier `kitchen` to be of type `struct on_off`:

```
struct on_off {  
    unsigned light : 1;  
    unsigned toaster : 1;  
    int count;          /* 4 bytes */  
    unsigned ac : 4;  
    unsigned : 4;  
    unsigned clock : 1;  
    unsigned : 0;  
    unsigned flag : 1;  
} kitchen ;
```

The structure `kitchen` contains eight members totalling 16 bytes. The following table describes the storage that each member occupies:

Member Name	Storage Occupied
light	1 bit
toaster	1 bit

(padding - 30 bits)	To next int boundary
count	The size of an int
ac	4 bits
(unnamed field)	4 bits
clock	1 bit
(padding - 23 bits)	To next int boundary (unnamed field)
flag	1 bit
(padding - 31 bits)	To next int boundary

All references to structure fields must be fully qualified. For instance, you cannot reference the second field by toaster. You must reference this field by kitchen.toaster. The following expression sets the light field to 1:

```
kitchen.light = 1;
```

When you assign to a bit field a value that is out of its range, the bit pattern is preserved and the appropriate bits are assigned. The following expression sets the toaster field of the kitchen structure to 0 because only the least significant bit is assigned to the toaster field:

```
kitchen.toaster = 2;
```