Bit-fields

Theory portion:

While we're on the subject of structures, we might as well look at bit fields. They can only be declared inside a structure or a union, and allow you to specify some very small objects of a given number of bits in length. Their usefulness is limited and they aren't seen in many programs, but we'll deal with them anyway. This example should help to make things clear:

**struct**

**{**

**/\* field 4 bits wide \*/**

**unsigned field1 :4;**

**/\***

**\* unnamed 3 bit field**

**\* unnamed fields allow for padding**

**\*/**

**unsigned        :3;**

**/\***

**\* one-bit field**

**\* can only be 0 or -1 in two's complement!**

**\*/**

**signed field2   :1;**

**/\* align next field on a storage unit \*/**

**unsigned        :0;**

**unsigned field3 :6;**

**}full\_of\_fields;**

Each field is accessed and manipulated as if it were an ordinary member of a structure. The keywords signed and unsigned mean what you would expect, except that it is interesting to note that a 1-bit signed field on a two's complement machine can only take the values 0 or -1. The declarations are permitted to include the const and volatile qualifiers.

The main use of bitfields is either to allow tight packing of data or to be able to specify the fields within some externally produced data files. C gives no guarantee of the ordering of fields within machine words, so if you do use them for the latter reason, you program will not only be non-portable, it will be compiler-dependent too. The Standard says that fields are packed into ‘storage units’, which are typically machine words. The packing order, and whether or not a bitfield may cross a storage unit boundary, are implementation defined. To force alignment to a storage unit boundary, a zero width field is used before the one that you want to have aligned.

Be careful using them. It can require a surprising amount of run-time code to manipulate these things and you can end up using more space than they save.

Bit fields do not have addresses—you can't have pointers to them or arrays of them.

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

struct

{

unsigned int widthValidated;

unsigned int heightValidated;

} 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 situation. 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 numbers of bytes. For example, above structure can be re-written as follows:

struct

{

unsigned int widthValidated : 1;

unsigned int heightValidated : 1;

} status;

Now, the above structure will require 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 status structure will use 4 bytes, but as soon as you will have 33 variables, then it will allocate next slot of the memory and it will start using 8 bytes. Let us check the following example to understand the concept:

#include <stdio.h>

#include <string.h>

/\* define simple structure \*/

struct

{

unsigned int widthValidated;

unsigned int heightValidated;

} status1;

/\* define a structure with bit fields \*/

struct

{

unsigned int widthValidated : 1;

unsigned int heightValidated : 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 form inside a structure:

**struct**

**{**

**type [member\_name] : width ;**

**};**

Below the description of variable elements of a bit field:

|  |  |
| --- | --- |
| Elements | Description |
| type | An integer type that determines how the bit-field's value is interpreted. The type may be int, signed int, unsigned int. |
| member\_name | The name of the bit-field. |
| 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 only then you can define a bit field with a width of 3 bits as follows:

**struct**

**{**

**unsigned int age : 3;**

**} Age;**

The above structure definition instructs C compiler that age variable is going to use only 3 bits to store the value, **if you will try to use more than 3 bits then it will not allow you to do so. Let us try the following example:**

**#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;**

**}**

When the above code is compiled it will compile with warning **(large integer implicitly truncated to unsigned type)**and when executed, it produces the following result:

**(However, there is no compilation error. There is warning.)**

Sizeof( Age ) : 4

Age.age : 4

Age.age : 7

Age.age : 0

**#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 = 9;**

**printf( "Age.age : %d\n", Age.age );**

**return 0;**

**}**

Sizeof( Age ) : 4

Age.age : 4

Age.age : 7

Age.age : 1

**(this will basically print given value % maximum value in case of unsigned range.)**

MCQ Portion:

**1.What is the correct syntax to initialize bit-fields in a structure?** a) struct temp  
 {  
 unsigned int a : 1;

}s;  
 b) struct temp  
    {  
 unsigned int a = 1;  
     }s;  
 c) struct temp  
     {  
 unsigned float a : 1;

}s;  
 d) Both a and c.

Answer is a)

However, bit fields cannot be of floating type

**2.Which of the following data types are accepted while declaring bit-fields?** a) char  
 b) float  
 c) double  
 d) None of the mentioned

Answer is a) char

**3.Which of the following reduces the size of a structure?** a) union  
 b) bit-fields  
 c) malloc  
 d) None of the mentioned

Answer is b) bit-fields

4.For what minimum value of x in a 32-bit Linux OS would make the size of s equal to 8 bytes?

1. struct temp
2. {
3. int a : 13;
4. int b : 8;
5. int c : x;
6. }s;

a) 4  
 b) 8  
 c) 12  
 d) 32

x upto 11 will limit the structure size in memory as 4 bytes but x=12 would make the structure size in memory as 8 bytes

So, answer is c) 12

5.(Assuming size of int = 4, calculate the % using the memory that would be occupied without bit-fields)

1. struct temp
2. {
3. int a : 1;
4. int b : 2;
5. int c : 4;
6. int d : 4;
7. }s;

a) 25%  
 b) 33.3%  
 c) 50%  
 d) 75%

Now, without bit field the size of the structure would be 16 bytes but after using bitfields the structure would take 4 bytes in memory space.

**6.In the declaration of bit-fields,** struct-declarator:  
 declarator  
 type-specifier declarator opt : constant-expression  
  
 The constant-expression specifies

a) The width of the field in bits.  
 b) Nothing  
 c) The width of the field in bytes.  
 d) Error

Answer is a) The width of the field in bits

**7. In the declaration of bit-fields,**  
 struct-declarator:

declarator  
 type-specifier declarator opt : constant-expression  
   
The constant-expression must be  
 a) Any type  
 b) Nothing  
 c) Integer value  
 d) Nonnegative integer value

Answer is d) Nonnegative integer value

**8.Which of the following is not allowed?**  
 a) Arrays of bit fields  
 b) Pointers to bit fields  
 c) Functions returning bit fields  
 d) None of the mentioned

Answer is d) None of the mentioned

**9.Bit fields can only be declared as part of a structure.**  
 a) false  
 b) true  
 c) Nothing  
 d) Varies

Answer is given as b) true. (But, as if I learnt that bitfields can also be part of union)

**10.The following declarations in order are**  
 short a : 17;  
 int long y : 33;  
  
 a) Legal, legal  
 b) Legal, illegal  
 c) Illegal, illegal  
 d) Illegal, legal

Answer is given as c) 

**11.What is the output of this C code?**

1. #include <stdio.h>
2. struct p
3. {
4. char x : 2;
5. int y : 2;
6. };
7. int main()
8. {
9. struct p p;
10. p.x = 2;
11. p.y = 1;
12. p.x = p.x & p.y;
13. printf("%d\n", p.x);
14. return 0;
15. }

a) 0  
 b) Compile time error  
 c) Undefined behaviour  
 d) Depends on the standard.

Answer is a) 0

Here, both x and y members have same width

12.What is the output of this C code?

1. #include <stdio.h>
2. union u
3. {
4. struct p
5. {
6. unsigned char x : 2;
7. unsigned int y : 2;
8. };
9. int x;
10. };
11. int main()
12. {
13. union u u;
14. u.p.x = 2;
15. printf("%d\n", u.p.x);
16. return  0;
17. }

a) Compile time error  
 b) Undefined behaviour  
 c) Depends on the standard  
 d) 2

Because, p is here a new structure type, not a variable of structure type. So, when we will try to initialize 2 to u.p.x or will try to print u.p.x's value compiler would say ‘union u’ has no member named ‘p’

However, the following code would not generate any compilation error

1. #include <stdio.h>
2. union u
3. {
4. struct name
5. {
6. unsigned char x : 2;
7. unsigned int y : 2;
8. }p;
9. int x;
10. };
11. int main()
12. {
13. union u u;
14. u.p.x = 2;
15. printf("%d\n", u.p.x);
16. return 0;
17. }

or this one: (this one also would not generate any compilation error. Rather, it would print 2)

1. #include <stdio.h>
2. union u
3. {
4. struct p
5. {
6. unsigned char x : 2;
7. unsigned int y : 2;
8. };
9. int x;
10. };
11. int main()
12. {
13. union u u;
14. u.p.x = 2;
15. printf("%d\n", u.p.x);
16. return 0;
17. }

But this one would generate compilation error due to wrong initialization manner

13.What is the output of this C code?

1. #include <stdio.h>
2. union u
3. {
4. struct
5. {
6. unsigned char x : 2;
7. unsigned int y : 2;
8. }p;
9. int x;
10. };
11. int main()
12. {
13. union u u.p.x = 2;
14. printf("%d\n", u.p.x);
15. return 0;
16. }

**14.What is the output of this C code?**

1. #include <stdio.h>
2. union u
3. {
4. struct
5. {
6. unsigned char x : 2;
7. unsigned int y : 2;
8. }p;
9. int x;
10. };
11. int main()
12. {
13. union u u = {2};
14. printf("%d\n", u.p.x);
15. return 0;
16. }

a) Compile time error  
 b) 2  
 c) Depends on the standard  
 d) None of the mentioned

Answer is b) 2

**15.What is the output of this C code?**

1. #include <stdio.h>
2. union u
3. {
4. struct
5. {
6. unsigned char x : 2;
7. unsigned int y : 2;
8. }p;
9. int x;
10. };
11. int main()
12. {
13. union u u.p = {2};
14. printf("%d\n", u.p.x);
15. return 0;
16. }

a) Compile time error  
 b) 2  
 c) Undefined behaviour  
 d) None of the mentioned

Answer is a) Compilation error due to wrong initialization

16.What is the output of this C code?

1. #include <stdio.h>
2. struct p
3. {
4. unsigned int x : 2;
5. unsigned int y : 2;
6. };
7. int main()
8. {
9. struct p p;
10. p.x = 3;
11. p.y = 1;
12. printf("%d\n", sizeof(p));
13. return 0;
14. }

a) Compile time error  
 b) Depends on the compiler  
 c) 2  
 d) 4

Answer is d) 4

17.What is the output of this C code?

1. #include <stdio.h>
2. struct p
3. {
4. unsigned int x : 2;
5. unsigned int y : 2;
6. };
7. int main()
8. {
9. struct p p;
10. p.x = 4;
11. p.y = 1;
12. printf("%d\n", sizeof(p));
13. return 0;
14. }

a) Compile time error  
 b) Depends on the compiler  
 c) 2  
 d) 4

When we compile this code, this would generate warning as large integer implicitly truncated to unsigned type [-Woverflow]

in p.x = 4; statement

So, if we print p.x and p.y this time, it would print 0 and 1 respectively

18.What is the output of this C code?

1. #include <stdio.h>
2. struct p
3. {
4. unsigned int x : 2;
5. unsigned int y : 2;
6. };
7. int main()
8. {
9. struct p p;
10. p.x = 3;
11. p.y = 4;
12. printf("%d\n", p.y);
13. return 0;
14. }

a) 0  
 b) 4  
 c) Depends on the compiler  
 d) 2

Answer is a) 0 Since p.y has 2 bits of size and we are trying to initialize a greater value (according to size) to it. So, overflow takes place. And some necessary steps will be taken.

19.What is the output of this C code?

1. #include <stdio.h>
2. struct p
3. {
4. unsigned int x : 7;
5. unsigned int y : 2;
6. };
7. int main()
8. {
9. struct p p;
10. p.x = 110;
11. p.y = 2;
12. printf("%d\n", p.x);
13. return 0;
14. }

a) Compile time error  
 b) 110  
 c) Depends on the standard  
 d) None of the mentioned

Answer is b) 110

20.What is the output of this C code?

1. #include <stdio.h>
2. struct p
3. {
4. unsigned int x : 1;
5. unsigned int y : 1;
6. };
7. int main()
8. {
9. struct p p;
10. p.x = 1;
11. p.y = 2;
12. printf("%d\n", p.y);
13. return 0;
14. }

a) 1  
 b) 2  
 c) 0  
 d) Depends on the compile

Answer is c) 0

21.What is the output of this C code?

1. #include <stdio.h>
2. struct p
3. {
4. char x : 2;
5. int y : 2;
6. };
7. int main()
8. {
9. struct p p;
10. p.x = 4;
11. p.y = 1;
12. p.x = p.x & p.y;
13. printf("%d\n", p.x);
14. return 0;
15. }

a) 0  
 b) Compile time error  
 c) Undefined behaviour  
 d) Depends on the standard.

Answer is a) 0 (0 & 1=0)