Bit Fields in C

In C, we can specify size (in bits) of structure and union members. 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 withing a small range.

For example, consider the following declaration of date without use of bit fields.

```
#include <stdio.h>

// A simple representation of date
struct date
{
   unsigned int d;
   unsigned int m;
   unsigned int y;
};

int main()
{
   printf("Size of date is %d bytes\n", sizeof(struct date));
   struct date dt = {31, 12, 2014};
   printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
}
```

Run on IDE

Output:

```
Size of date is 12 bytes
Date is 31/12/2014
```

The above representation of 'date' takes 12 bytes on a compiler where an unsigned int takes 4 bytes. Since we know that the value of d is always from 1 to 31, value of m is from 1 to 12, we can optimize the space using bit fields.

```
#include <stdio.h>

// A space optimized representation of date
struct date
{
   // d has value between 1 and 31, so 5 bits
   // are sufficient
   unsigned int d: 5;

   // m has value between 1 and 12, so 4 bits
   // are sufficient
   unsigned int m: 4;

   unsigned int y;
};

int main()
{
   printf("Size of date is %d bytes\n", sizeof(struct date));
   struct date dt = {31, 12, 2014};
   printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
   return 0;
}
```

Run on IDE

Output:

```
Size of date is 8 bytes
Date is 31/12/2014
```

Following are some interesting facts about bit fields in C.

1) A special unnamed bit field of size 0 is used to force alignment on next boundary. For example consider the following program.

```
#include <stdio.h>

// A structure without forced alignment
struct test1
{
   unsigned int x: 5;
   unsigned int y: 8;
};

// A structure with forced alignment
struct test2
{
   unsigned int x: 5;
   unsigned int: 0;
   unsigned int y: 8;
};

int main()
{
   printf("Size of test1 is %d bytes\n", sizeof(struct test1));
   printf("Size of test2 is %d bytes\n", sizeof(struct test2));
   return 0;
}
```

Run on IDE

Output:

```
Size of test1 is 4 bytes
Size of test2 is 8 bytes
```

2) We cannot have pointers to bit field members as they may not start at a byte boundary.

```
#include <stdio.h>
struct test
{
   unsigned int x: 5;
   unsigned int y: 5;
   unsigned int z;
};
int main()
{
   struct test t;

   // Uncommenting the following line will make
   // the program compile and run
   printf("Address of t.x is %p", &t.x);

   // The below line works fine as z is not a
   // bit field member
   printf("Address of t.z is %p", &t.z);
   return 0;
}
```

Run on IDE

Output:

```
error: attempt to take address of bit-field structure member 'test::x'
```

3) It is implementation defined to assign an out-of-range value to a bit field member.

```
#include <stdio.h>
struct test
{
   unsigned int x: 2;
   unsigned int y: 2;
   unsigned int z: 2;
};
int main()
{
   struct test t;
   t.x = 5;
   printf("%d", t.x);
   return 0;
}
```

Run on IDE

Output:

```
Implementation-Dependent
```

4) In C++, we can have static members in a structure/class, but bit fields cannot be static.

```
// The below C++ program compiles and runs fine
struct test1 {
   static unsigned int x;
};
int main() {  }


// But below C++ program fails in compilation as bit fields
// cannot be static
struct test1 {
   static unsigned int x: 5;
};
int main() {  }
// error: static member 'x' cannot be a bit-field
```

Run on IDE

5) Array of bit fields is not allowed. For example, the below program fails in compilation.

```
struct test
{
  unsigned int x[10]: 5;
};

int main()
{
}
```

Run on IDE

Output:

```
error: bit-field 'x' has invalid type
```

Exercise:

Predict the output of following programs. Assume that unsigned int takes 4 bytes and long int takes 8 bytes.

1)

```
#include <stdio.h>
struct test
{
   unsigned int x;
   unsigned int y: 33;
   unsigned int z;
};
int main()
{
   printf("%d", sizeof(struct test));
   return 0;
}
```

Run on IDE

2)

```
#include <stdio.h>
struct test
{
   unsigned int x;
   long int y: 33;
   unsigned int z;
};
int main()
{
   struct test t;
   unsigned int *ptr1 = &t.x;
   unsigned int *ptr2 = &t.z;
   printf("%d", ptr2 - ptr1);
   return 0;
}
```

Run on IDE

3)

```
union test
{
  unsigned int x: 3;
  unsigned int y: 3;
  int z;
};

int main()
{
   union test t;
   t.x = 5;
   t.y = 4;
```

```
    t.z = 1;
    printf("t.x = %d, t.y = %d, t.z = %d",
           t.x, t.y, t.z);
    return 0;
}
```

Run on IDE

4) Use bit fields in C to figure out a way whether a machine is little endian or big endian.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Practice Tags : Bit Magic C

Article Tags : Bit Magic C C-Struct-Union-Enum cpp-structure

Login to Improve this Article

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.

Recommended Posts:

Structure Sorting (By Multiple Rules) in C++
Structure Member Alignment, Padding and Data Packing
Operations on struct variables in C
Memory Layout of C Programs
Struct Hack
Check if a number has same number of set and unset bits
Sum of all elements up to Nth row in a Pascal triangle
Check if concatenation of two strings is balanced or not
Find i'th index character in a binary string obtained after n iterations | Set 2
Sudo Placement | Range Queries

(Login to Rate)

3.3 Average Difficulty : **3.3/5.0**
Based on **36** vote(s)

□ Add to TODO List

□ Mark as DONE

Feedback Add Notes

Basic Easy Medium Hard Expert

Improve Article

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments Share this post!