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);
}
```

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;
}
```

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;
}
```

```
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.

```c
#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;
}
```

Output:

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

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

```c
#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;
}
```

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
```

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()
{

}
```

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)

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

```
prog.c:5:1: error: width of 'y' exceeds its type
 unsigned int y: 33;
 ^
```

2)

```c
#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;
}
```

```
4
```

3)

```c
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;
}
```

```
t.x = 1, t.y = 1, t.z = 1
```