2022

PRACTICAL WORK NO. 3 BIT FIELDS IN C

AUTOR: FACUNDO LAUTARO COSTARELLI
FILE: 176.291-6
TEACHER: MARIANO GONZALEZ MARTINEZ
SUBJECT: COMPUTER SCIENCE 1
COURSE: R1001

INTRODUCTION

When memory space is limited, the aim is to store several "**objects**" or "**pieces of information**" within a single **machine word**, where that word refers to a size of data or information that the CPU can handle optimally. That size is measured in bits and is usually associated with the size corresponding to the "**CPU registers and addresses".** The size of the word will depend on the architecture of the processor and also the compiler is responsible for handling this information and its optimization, so you only have to worry about occupying the memory efficiently without taking up unnecessary space.

So, information packaging, in a word, can be used in situations such as: symbol tables for compilers, interfaces to hardware devices, external data formats, etc.

In particular, if we treat data packaging through the example of the symbol table manipulated by a compiler, we see that each "**identifier or name**" of a macro or variable within a program code has particular associated information such as: whether it is reserved or not, whether it is external or static, etc. One way to "**compact or package**" these features for an identifier is through a bit-sized "**set of flags**" with that bit being within the range of "char or int". For this, ma1cros are used, now called "**masks**" that will have constant values associated with the starting positions of the groups of bits of information that represent these masks. This can be written as:

#dfine KEYWORD 01

#define EXTERNAL 02

#define STATIC 04

Or also as:

enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 };

The numbers 01, 02, 04, etc., must all be powers of 2. Now, in order to access these groups of bits of information associated with masks and combine their use with flags, bit operators such as shift operators, masking, complement, etc. must be used. See the following example:

```
flags |= EXTERNAL | STATIC;
```

enciende los bits EXTERNAL y STATIC en flags, en tanto que

```
flags &= ~(EXTERNAL | STATIC);
```

los apaga, y

```
if ((flags & (EXTERNAL | STATIC)) == 0) ...
```

es verdadero si ambos bits están apagados.

However, writing in this way is no longer so common and is also confusing, so the C language offers the alternative of packaging information into groups of bits and accessing it much more easily and directly than using logical operators. This is through the concept of "**bit fields**" which is used through structures and junctions providing simplicity.

BIT FIELDS

A **bit field** is an adjacent or contiguous set of bits, located in a storage unit or memory space which is assigned by the Operating System once the use of the bit field in question is implemented or defined. This group of bits will now be associated with the concept of "word" and will represent a "machine word". The C syntax for defining and accessing bit fields is based on structures and junctions. For example, the above symbol table can be rewritten as:

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern  : 1;
    unsigned int is_static  : 1;
} flags;
```

We see that a variable with the identifier "flags" is defined associated with the "struct" data type. This variable is then a structure with 3 members where each of them has information that can be represented with only 1 bit per member for this example. We see that after declaring each type of data of each member and the identifier of the member in question, " : " is written after the name of the member and then a constant value e.g. "1" associated with the bit size of the information stored by that member. To

access the members now called fields, you access in the same way as in an ordinary structure, that is, in this case: flags.is__keyword, flag.is__extern, etc.

Bit fields are said to behave like "small integers" so you can perform arithmetic operations as with any integer. For example:

flags.is__extern = flags.is__static = 1; Syntax for turning on the bits

flgas.is__extern = flags.is__static = 0; Syntax for turning off bits

if(flag.is__extern == 0 & flags.is__static == 0) //syntax for evaluating condition

This allows more operations to be carried out.

The bit fields are assigned from left to right or from right to left by some computers, so you have to see well how this happens in each machine, especially when working with data that is defined externally, since a group of bits, when working in one direction or another, can mean one type of information or another. The codes that depend on this situation are not portable. In addition, some computers allow bit fields to "cross" or "exceed" word boundaries, while others do not. Bit fields can only be declared as "integers" and the possibilities of "unsigned int" and "singed int" must be made explicit to ensure portability of the code. Although other types of data such as "long, unsigned char, etc" can also be used. So almost everything about the bit fields is dependent on the implantation that is given to it, that is, it is concluded that the manipulations of the bit fields are dependent on the machine.

Also, it's important to note that bit fields are not arrays or arrays of bits, or pointers, etc. It is also not possible to work with individual bits, the direction of each bit, or the direction of the bit field, so the "&" operator cannot be used. Since the bit field is considered as the member of a structure or union and to access these fields, they are worked as members. A bit field is said to have no address.

Advantages of bit fields: Although so far it has been explained what bit fields are, how they are used, the syntax, and a simple example, it arises to ask: What are the advantages of using bit fields? So we can answer the following: Bit fields are used for maintenance of internal data structures

and to use memory more efficiently since the data is stored with the minimum number of bits possible without taking up unnecessary additional space. This is extremely applicable and useful when we know in advance that a value of a member of a structure or union does not exceed a limit value that it can reach, or even when that value is within a small range of values that are all representable with the same number of minimum bits. A full example for signed and unsigned integer data will be shown later in this report.

Disadvantages of bitfields: The use of bitfields causes the compiler to generate code in machine language such that it runs slower. This is due to having access to only portions of an addressable memory space, which takes more machine language operations.

Example bit field for unsigned integers:

In principle, let's look at the following simple code with its output for the case where no bit field is used but could be used. In this code, what is loaded into the structure and printed in the output are the values related to the day, month and year, all unsigned integer values.

```c
#include <stdio.h>
// A simple representation of the date
struct date {
    unsigned int d;
    unsigned int m;
    unsigned int y;
};

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

On-screen output:

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

We see that the structure being declared as it is with its members, weighs 12 bytes for a compiler and 32-bit architecture machine where each data of the unsigned int type weighs 4 bytes. In this example, you can see that we won't be using the entire range of values that an unsigned int data type covers, but only a few. For example, "days or dates" range from 1 to

31 as a maximum of unsigned integers, "months or months" range from 1 to 12 as a maximum of unsigned integers, "years or years" could cover a range from 2014 to 2100 as a maximum according to the maximum statistical life span of a person, again unsigned integer values. These values do not necessarily need to be represented by all the 32 bits that make up the 4 bytes but only by a "group of bits"

For example, an unsigned integer value such as 4, associated for example with the day number of a given month, can be represented with 3 bits, i.e. almost half a byte. However, when declaring a data as "unsigned int" it will occupy 4 bytes or 8 bytes according to 32 or 64-bit computer architecture, this indicates that said unsigned integer data of 4 is taking up more space than necessary at the level of number of bits, that is, its representation would be "0000000000.....000100" where the equivalence of 4 reaches the representation of "100" while the extra 0 in front of this trio of bits, are filled or added by the OS taking up additional memory space. To avoid this, we use bit fields such that, in the example, the unsigned integer 4 has only the minimum bits necessary for its representation associated with it, i.e. 3 bits in this example thus occupying less static memory. This analysis can be repeated for values that may take months and years.

So, if we apply a bit field to speed up memory, we have the following example:

```c
#include <stdio.h>
// A simple representation of the date
struct date {
    unsigned int d : 5;
    unsigned int m : 4;
    unsigned int y : 12;
};

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

On-screen output:

```
Size of date is 4 bytes
Date is 31/12/2014
```

From this code we see that:

- 5 bits are enough to represent all the unsigned integer values associated with a date, including the worst case, which would be 31.

- 4 bits are enough to represent all the unsigned integer values associated with a month including the worst case which would be 12

- 12 bits are enough to represent all the unsigned integer values associated with a year, including the worst case, which would be 2100.

We see that the "date" structure now weighs 4 bytes and not 12, that is, 8 bytes less than before. If this is applied to extensive multi-line code running on a small computer with limited resources, including the amount of memory, then it can be concluded that several bytes and megabytes can be saved if necessary.

Example bit field for signed integers:

In principle, let's look at the following simple code with its output for the case where a bit field is not used, but could be used. In this code, what is loaded into the structure and printed in the output are the values related to the day, month and year, all signed integer values.

```c
#include <stdio.h>
// A simple representation of the date
struct date {
     int d;
     int m;
     int y;
};

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

On-screen output:

```
Size of date is 4 bytes
Date is 31/12/2014
```

For this example I take the previous code, but now modifying the declaration of the structure and the type of data of each member, all of them being integer variables with signs. We see that the structure being declared as it is with its members, weighs 12 bytes for a compiler and machine architecture of 32 bits where each data of the int type weighs 4 bytes. If we carry out the same analysis as in the previous example, we will reach the same conclusions regarding the efficient use of memory, so:

- 5 bits are enough to represent all the signed integer values associated with a date, including the worst case, which would be 31.

- 4 bits are enough to represent all the signed integer values associated with a month including the worst case which would be 12

- 12 bits are enough to be able to represent all the signed integer values associated with a year including the worst case which would be 2100

So, if we apply a bit field to speed up memory, we have the following example:

```c
#include <stdio.h>

// Space optimized representation of the date
struct date {
    int d : 5;
    int m : 4;
    int y : 12;
};

int main()
{
    printf("Size of date is %lu 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;
}
```

On-screen output:

```
Size of date is 8 bytes
Date is -1/-4/2014
```

We see that something unexpected is happening at first. The value of the day and the month are different from those charged and are also negative. If we take as analysis the expected value 31 of the day, it was statically stored in 5 bits of signed integer, binarily regardless of the type of integer at first it is assigned a representation of "11111" which in principle seems to be fine. The most significant bit is "1" and since it is working with a signed integer, the machine now considers the binary representation as negative due to the presence of the MSB = 1, in this case this representation if we were to reconvert it back to a signed integer would be "-15" integer; Then to solve this, the machine converts said binary expression to the corresponding positive one in order to correctly associate the positive 31 where this number was previously recognized and loaded as +31 with a binary expression according to the type of integer data signed. The process it performs is the complement to 2 and it does it internally, then it gets "00001" and displays the associated representation on the screen in a signed integer. The expected would be that the machine would show on its own that this number is worth "31"

but in reality it will show "-1" since when declaring signed January data, the binary representation "00001" by table and convention is worth "-1.". The same happens with the value of the month charged as 12 which is now worth -4 at the exit. This does not happen with the value of the year 2014 since its binary representation has MSB = 0 so it is already positive.

To solve these cases where a different number is returned and with an opposite sign to the expected one, it would be necessary to make a code or function that somehow reconverts the binary representation calculated by the machine in such a way that a new binary representation is obtained and associated with the value of the integer entered. Another easier option is to use signed integers for negative values and unsigned integers for positive values.

Clarification of the examples: The examples presented are the simplest to understand and see in bitfield applications. In particular, it should be noted that bit fields are commonly used with struct structures, but also with unions. In the following special cases we will evaluate both structures and joints.

Special cases of bit fields:

1) Bit field without identifier or name with bit 0:

An example is:

```c
#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 %lu bytes\n",
           sizeof(struct test1));
    printf("Size of test2 is %lu bytes\n",
           sizeof(struct test2));
    return 0;
}
```

On-screen output:

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

This is used to perform a mandatory "alignment" between the current bit field and the next boundary bit field. Another way to say it is, "The special width 0 can be used to force the alignment to the next word boundary." We see that variable **data** alignment is associated with the way data is stored in memory banks. For example, the natural alignment of int data on a 32-bit machine is 4 bytes, this implies that the data read start offset is placed or jumped from 4 byte blocks where between block and block there is information that is read. It is said that when a type of data is naturally aligned, the CPU reads it in minimal read machine cycles.

2) Bit field with pointers:

An example is:

```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;
}
```

On-screen output:

```
prog.c: In function 'main':
prog.c:14:1: error: cannot take address of bit-field 'x'
 printf("Address of t.x is %p", &t.x);
 ^
```

As stated earlier in this report, you cannot use pointers to member bit fields of structures or joins because such fields do not necessarily start at the "start byte" associated with the machine's read and/or write alignment. If you want to use this, errors are thrown.

3) Bit field with out-of-range value assignment:

An example is:

```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;
}
```

On-screen output:

```
Implementation-Dependent
```

We see that, at the moment of wanting to assign an integer value of any type, in this case unsigned, which is "outside the binary representation range", that is, a value for which more bits are needed to represent it than the number of bits set by the programmer in the bit field declaration. In this example, the situation is given for the bit field "x" of the structure "test". As mentioned in this report, the possibility of a bit field "crossing" or "exceeding" the word limit that was defined by the programmer in the declaration of that field depends exclusively on the machine on which the program is running, since some allow it and others do not. That is, it is "implementation dependent", which can be seen in the printed output on the screen.

4) Bit field with arrays:

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

int main()
{
}
```

On-screen output:

```
prog.c:3:1: error: bit-field 'x' has invalid type
 unsigned int x[10]: 5;
 ^
```

When we want to declare and use a bit field as an "array" of bits, we see that as said before, this is not possible and an error is thrown. Both for the use of bit field in structures and joints. This is because each bit does not have a specified start address, there are only addresses defined for bytes and groups of bytes according to the alignment of the computer and the types of data that exist in the language.

5) Bit field with bonds:

It is useful for defining Boolean variables, thus saving memory.

```
union {
  struct {
    unsigned int waitingSomeEvent: 1;
    unsigned int ready4something : 1;
    ......
    unsigned int someEvent       : 1;
    unsigned int anotherFlag     : 1;
  };
  unsigned int flags;
};
```

- Poner a cero todos los flags:

```
flags = 0;
```

- Modificar o testear algún flag:

```
if( waitingSomeEvent && someEvent )
{
     someEvent = 0;
}
```

CONCLUSIONS

This report allowed us to emphasize more on the concepts of bit fields and obtain multiple application examples, thus concluding that they are useful when wanting to reduce memory consumption significantly without losing data and knowing in advance the values that the data will take and the fair and necessary sizes that they will occupy in memory

BIBLIOGRAPHY

https://www.geeksforgeeks.org/bit-fields-c/

https://www.dsi.fceia.unr.edu.ar/images/Estructuras_clase2_2020.pdf

https://es.acervolima.com/alineacion-relleno-y-empaquetado-de-datos-de-miembros-de-estructura/

How to program in C/C++ 2nd Edition. H. M Detiel and J.P Deitel. Chapter 6 Section 6.9

The C Programming Language – 2nd Edition. Brian W. Kernighan and Dennis M. Ritchie. Chapter 10 Section 10.6