2022

TRABAJO PRACTICO NRO 3
CAMPOS DE BITS EN C

AUTOR: FACUNDO LAUTARO COSTARELLI
LEGAJO: 176.291-6
PROFESOR: MARIANO GONZALEZ MARTINEZ
MATERIA: INFORMATICA 1
CURSO: R1001

INTRODUCCIÓN

Cuando el espacio en memoria es reducido, se busca guardar varios "**objetos**" o "**pedazos de información**" dentro de una sola **palabra de máquina**, donde dicha palabra refiere a un tamaño de data o información que el CPU puede manejar óptimamente. Ese tamaño es medido en bits y suele estar asociado ese tamaño con el tamaño correspondiente a los "**registros y direcciones del CPU**". El tamaño de la palabra dependerá de la arquitectura del procesador y además el compilador se encarga del manejo de esta información y su optimización por lo que solamente hay que preocuparse en ocupar la memoria eficientemente sin ocupar espacios innecesarios.

Entonces, el empaquetar información, en una palabra, puede ser usado en situaciones como: tablas de símbolos para compiladores, interfaces hacia dispositivos de hardware, formatos de datos externos, etc.

En particular, si tratamos el empaquetamiento de data a través del ejemplo de la tabla de símbolo manipulada por un compilador, vemos que cada "**identificador o nombre**" de una macro o variable dentro de un código de programa tiene información particular asociada como por ej: si está reservada o no, si es externo o estático, etc. Una forma de "**compactar o empaquetar**" estas características para un identificador, es a través de un "**conjunto de banderas o flags**" de tamaño de un bit siendo ese bit dentro del rango de "char o int". Para esto, se utilizan ma1cros llamadas ahora "**mascaras**" que tendrán valores constantes asociados a las posiciones de comienzo de los grupos de bits de información que representan dichas mascaras. Esto se puede escribir como:

```
#dfine KEYWORD 01

#define EXTERNAL 02

#define STATIC 04
```

O también como:

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

Los números 01, 02, 04, etc, deben ser todos potencias de 2. Ahora, para poder acceder a estos grupos de bits de información asociados a las máscaras y combinar su uso con flags, se deben usar operadores de bits como por ej: operadores de corrimiento, enmascaramiento, complemento, etc. Véase el siguiente ejemplo:

```
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.

Sin embargo, escribir de esta forma ya no es tan usual y además es confusa, por lo que el lenguaje C ofrece la alternativa de empaquetar la información en grupos de bits y acceder a ella de forma mucho más fácil y directa que usando operadores lógicos. Esto es a través del concepto de "**campos de bits**" lo cual se usa a través de estructuras y uniones brindando simplicidad.

CAMPOS DE BITS

Un **campo de bits** es un conjunto adyacente o contiguo de bits, ubicado este conjunto en una unidad de almacenamiento o espacio de memoria el cual es asignado por el Sistema Operativo una vez que se implementa o se define el uso del campo de bits en cuestión. Este grupo de bits lo asociaremos ahora con el concepto de "palabra" y representará una "palabra de maquina". La sintaxis en C para la definición y acceso a los campos de bits, se base en estructuras y uniones. Por ejemplo, la tabla de símbolos anterior se puede reescribir como:

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

Vemos que se define una variable con identificador “flags” asociada al tipo de dato “struct”. Dicha variable es entonces una estructura con 3 miembros donde cada uno de ellos posee información que se puede representar con solamente 1 bit por miembro para este ejemplo. Vemos que luego de declarar cada tipo de dato de cada miembro y el identificador del miembro en cuestión, se escribe “ : “ a continuación del nombre del miembro y luego un valor constante ej “1” asociado al tamaño en bits de la información que almacena ese miembro. Para acceder a los miembros llamados ahora campos, se accede de la misma forma que a en una estructura común y corriente, es decir, en este caso: flags.is__keyword, flag.is__extern, etc.

Se dice que los campos de bits se comportan como “pequeños enteros” por lo que se pueden realizar operaciones aritméticas como con cualquier entero. Por ejemplo:

flags.is__extern = flags.is__static = 1; // sintaxis para encender los bits

flgas.is__extern = flags.is__static = 0; //sintaxis para apagar los bits

if(flag.is__extern == 0 && flags.is__static == 0) //sintaxis para evaluar condicion

Asi se pueden realizar más operaciones.

Los campos de bits se asignan de izquierda a derecha o de derecha a izquierda por parte de algunas computadoras por lo que hay que ver bien como sucede esto en cada maquina especialmente al momento de trabajar con datos que vienen definidos externamente ya que un grupo de bits al trabajarlo en un sentido u otro puede significar un tipo de información u otra. Los códigos que dependen de esta situación no son portables. Además, algunas computadoras permiten que los campos de bits “crucen” o “excedan” límites de palabras, mientras que otras no lo permiten. Los campos de bits solo pueden ser declarados como “enteros” siendo las posibilidades de “unsigned int” y “singed int” las cuales se deben explicitar, esto para asegurar portabilidad al código. Aunque también se pueden usar otros tipos de datos como “long, unsigned char, etc”. Por lo que casi todo acerca de los campos de bits es dependiente de la implantación que se le dé, es decir, se concluye que las manipulaciones de los campos de bits son dependientes de la máquina.

Además, es importante notar que los campos de bits no son arreglos o arrays de bits, ni punteros, etc. No se puede tampoco trabajar ni con bits individuales ni con la dirección de cada bit ni con la dirección del campo de bit por lo que el operador ”&” no se puede usar. Ya que el campo de bit es considerado como el miembro de una estructura o unión y para acceder a estos campos, se los trabajan como miembros. Se dice que un campo de bits no tiene dirección.

Ventajas de campos de bits: Si bien hasta ahora se explicó que son los campos de bits, como se usan, la sintaxis, y un ejemplo sencillo, surge preguntarse: ¿Cuáles son las ventajas del uso de campos de bits? Por lo que podemos responder lo siguiente: Los campos de bits son usados para mantenimiento de estructuras de datos internas y para usar de manera más eficiente la memoria ya que los datos son almacenados con el mínimo número de bits posible sin ocupar espacio adicional innecesario. Esto es sumamente aplicable y útil, cuando sabemos de antemano que un valor de un miembro de una estructura o unión, no excede un valor limite que puede alcanzar o mismo cuando dicho valor e encuentra dentro de un rango pequeño de valores todos representables con la misma cantidad de bits mínimos. Mas adelante en este informe se mostrará un ejemplo completo para datos enteros signados y sin signo.

Desventajas de campos de bits: El uso de campos de bits hace que el compilador genere código en lenguaje de maquina tal que este se ejecuta mas lento. Esto es debido al tener acceso a solo porciones de un espacio de memoria direccionable, lo cual toma más operaciones en lenguaje de máquina.

Ejemplo campo de bits para enteros sin signo:

En principio veamos el siguiente código sencillo con su salida para el caso en donde no se usan campo de bits pero se podrían usar. En este código lo que se carga en la estructura y se imprime en la salida son los valores relacionados al día, mes y año todos valores enteros sin signo.

```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);
}
```

Salida en pantalla:

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

Vemos que la estructura siendo declarada tal y como esta con sus miembros, pesa 12 bytes para un compilador y maquina de arquitectura de 32 bits donde cada dato del tipo unsigned int pesa 4 bytes. En este ejemplo, se puede ver que no usaremos todo el rango de valores que abarca un tipo de dato unsigned int sino solamente algunos. Por ejemplo, los “días o dates” van del rango 1 a 31 como máximo de enteros sin signo, los “meses o months” van del rango 1 a 12 como máximo de enteros sin signo, los “años o years” podría abarcar un rango desde el 2014 al 2100 como máximo acorde al periodo de vida máximo estadístico de una persona, nuevamente valores enteros sin signo. Estos valores, no necesariamente necesitan ser representados por todos los 32 bits que conforman los 4 bytes sino solamente por un “grupo de bits”

Por ejemplo, un valor entero sin signo como el 4, asociado por ejemplo al nro de día de un determinado mes, puede ser representado con 3 bits, , es decir, casi medio byte. Sin embargo, al declarar un dato como “unsigned int” este ocupara 4 bytes u 8 bytes según arquitectura de computadora de 32 o 64 bits, esto nos indica que dicho dato entero sin signo de 4 está ocupando más espacio del necesario a nivel de cantidad bits, Es decir su representación estaría siendo “000000000…..000100” donde la equivalencia del 4 alcanza con la representación de “100” mientras que los 0 extra delante de este trio de bits, son rellenados o agregados por el SO ocupando espacio de memoria adicional. Para evitar esto, se utiliza campos de bits tal que, del ejemplo, el 4 entero sin signo tiene solamente

asociado los bits mínimos necesarios para su representación, ósea 3 bits en este ejemplo ocupando así menos memoria estática. Este análisis se puede repetir para los valores que pueden tomar los meses y los años.

Entonces, si aplicamos campo de bits para agilizar la memoria tenemos el siguiente ejemplo:

```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);
}
```

Salida en pantalla:

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

De este código vemos que:

- 5 bit son suficientes para poder representar todos los valores enteros sin signo asociados a una fecha incluyendo el peor caso que sería 31.

- 4 bit son suficientes para poder representar todos los valores enteros sin signo asociados a un mes incluyendo el peor caso que seria 12

- 12 bit son suficientes para poder representar todos los valores enteros sin signo asociados a un año incluyendo el peor caso que seria 2100.

Vemos que la estructura “date” pesa ahora 4 bytes y no 12, ósea 8 bytes menos que antes. Si esto se aplica a un código extenso de múltiples líneas que corre en una pequeña computadora con recursos limitados, entre ellos, la cantidad de memoria, entonces se puede concluir que se pueden ahorrar varios bytes y megabytes de ser necesario.

Ejemplo campo de bits para enteros con signo:

En principio veamos el siguiente código sencillo con su salida para el caso en donde no se usan campo de bits, pero se podrían usar. En este código lo que se carga en la estructura y se imprime en la salida son los valores relacionados al día, mes y año todos valores enteros con signo.

```
#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);
}
```

Salida en pantalla:

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

Para este ejemplo tomo el código anterior, pero modificando ahora la declaración de la estructura y el tipo de dato de cada miembro siendo todos ellos variables enteras con signo. Vemos que la estructura siendo declarada tal y como esta con sus miembros, pesa 12 bytes para un compilador y máquina de arquitectura de 32 bits donde cada dato del tipo int pesa 4 bytes. Si realizamos el mismo análisis que en el ejemplo anterior llegaremos a las mismas conclusiones en cuanto a la utilización eficiente de la memoria, por lo que:

- 5 bit son suficientes para poder representar todos los valores enteros con signo asociados a una fecha incluyendo el peor caso que sería 31.

- 4 bit son suficientes para poder representar todos los valores enteros con signo asociados a un mes incluyendo el peor caso que seria 12

- 12 bit son suficientes para poder representar todos los valores enteros con signo asociados a un año incluyendo el peor caso que seria 2100

Entonces, si aplicamos campo de bits para agilizar la memoria tenemos el siguiente ejemplo:

```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;
}
```

Salida en pantalla:

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

Vemos que sucede algo en principio inesperado. El valor del día y del mes son distintos a los cargados y además están negativos. Si tomamos como análisis el valor 31 esperado del día, este se almaceno estáticamente en 5 bits de entero signado, binariamente independientemente del tipo de entero en un principio se le asigna una representación de “11111” que en principio parece estar bien. El bit más significativo es “1” y al estar trabajando con entero signado, la maquina considera ahora la

representación binaria como negativa por la presencia del MSB = 1, en este caso esta representación si la reconvirtiésemos devuelta a entero signado seria “-15” entero; Entonces para solucionar esto, la maquina convierte dicha expresión binaria a la correspondiente positiva para poder asociar correctamente al 31 positivo donde este número fue previamente reconocido y cargado como +31 con una expresión binaria acorde al tipo de dato entero signado. El proceso que realiza es el complemento a 2 y lo hace internamente, luego obtiene “00001” y muestra en pantalla la representación asociada en entero signado. Lo esperado seria que la maquina mostrase por si sola que ese nro vale “31” pero en realidad mostrara “-1” ya que al declarar dato enero signado, la representación binaria “00001” por tabla y convención vale “-1.”. Lo mismo sucede con el valor del mes cargado como 12 que ahora a la salida vale -4. No sucede esto con el valor del año 2014 ya que su representación binaria tiene MSB = 0 entonces ya está positivizada.

Para solucionar estos casos donde se devuelve un numero distinto y con signo opuesto al esperado, habría que realizar un código o función que reconvierta de alguna forma la representación binaria calculada por la maquina tal que se obtenga una nueva representación binaria y se asocie al valor del entero ingresado. Otra opción más fácil, es usar entero signado para valores negativos y enteros sin signar para valores positivos.

Aclaracion de los ejemplos: Los ejemplos planteados son los mas sencillos de entender y ver las aplicaciones de campo de bits. En particular cabe destacar que los campos de bits se usan habitualmente con estructuras tipo “struct” aunque también con “uniones” tipo “union”. En los siguientes casos especiales evaluaremos tanto con estructuras como uniones.

Casos especiales de campos de bits:

1) Campo de bit sin identificador o nombre con bit 0:

Un ejemplo es:

```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;
}
```

Salida en pantalla:

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

Esto se utiliza para realizar una “alineación” obligatoria entre el campo de bit actual y el campo de bit limite siguiente. Otra forma de decirlo es: “El ancho especial 0 puede emplearse para obligar a la alineación al siguiente límite de palabra”. Vemos que la **alineación** de datos de variables está asociada con la forma en que los datos se almacenan los bancos de memoria. Por ejemplo, la alineación natural de datos tipo int en una máquina de 32 bits es de 4 bytes, esto implica que el offset de comienzo de lectura de datos se coloca o salta de a bloques de 4 bytes donde entre bloque y bloque hay información que se lee. Se dice que cuando un tipo de datos está alineado de forma natural, la CPU lo lee en ciclos de máquina de lectura mínimos.

2) Campo de bits con punteros:

Un ejemplo es:

```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;
}
```

Salida en pantalla:

```
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);
 ^
```

Como se dijo antes en este informe, no se puede utilizar punteros a campos de bits miembros de estructuras o uniones ya que dichos campos no necesariamente empiezan en el “byte de comienzo” asociado a la alineación de lectura y/o escritura de la máquina. En caso de querer usar esto, se arrojan errores.

3) Campo de bit con asignación de valor fuera de rango:

Un ejemplo es:

```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;
}
```

Salida en pantalla:

```
Implementation-Dependent
```

Vemos que, al momento de querer asignar un valor entero de cualquier tipo, en este caso no signado, el cual se encuentra “fuera del rango de representación binaria”, es decir, un valor para el cual se necesitan mas bits para representarlo que la cantidad de bits fijadas por el programador en la declaración del campo de bit. En este ejemplo, la situación se da para el campo de bit “x” de la estructura “test” . Como se mencionó en este informe, la posibilidad de que un campo de bit “cruce” o “exceda” el límite de palabra que fue definido por el programador en la declaración de dicho campo, depende exclusivamente de la maquina en donde este corriendo el programa, ya que algunas lo permiten y otras no. Es decir, es “dependiente de la implementación”, lo cual se puede ver en la salida impresa en pantalla.

4) Campo de bit con arreglos:

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

int main()
{
}
```

Salida en pantalla:

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

Al momento de querer declarar y utilizar un campo de bit como un "arreglo" de bits, vemos que como se dijo antes, esto no es posible y se arroja un error. Tanto para el uso de campo de bits en estructuras y uniones. Esto es debido a que cada bit no tiene una dirección de comienzo especificada, solo existen direcciones definidas para bytes y grupos de bytes de acuerdo a la alineación de la computadora y a los tipos de dato que existen del lenguaje.

5) Campo de bit con uniones:

Es de utilidad para definir variables booleanas ahorrando así memoria.

```
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;
}
```

CONCLUSIONES

Este informe permitió enfatizar mas en los conceptos de campos de bits y obtener múltiples ejemplos de aplicación llegando así a la conclusión de que son de utilidad al momento de querer reducir el consumo de memoria significativamente sin perder datos y conociendo con anterioridad los valores que los datos tomarán y los tamaños justos y necesarios que ocuparán en memoria

BIBLIOGRAFIA

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/

Como programar en C/C++ 2da Edición. H. M Detiel y J.P Deitel. Capitulo 6 Sección 6.9

El lenguaje de programación C – 2da Edición. Brian W. Kernighan y Dennis M. Ritchie. Capítulo 10 Sección 10.6