

PROGRAMACIÓN EN BAJO NIVEL

AGENDA:

- Comunicación directa con el hardware
- Operadores a nivel de bit
- Campos de bits

¿Programación en bajo Nivel?

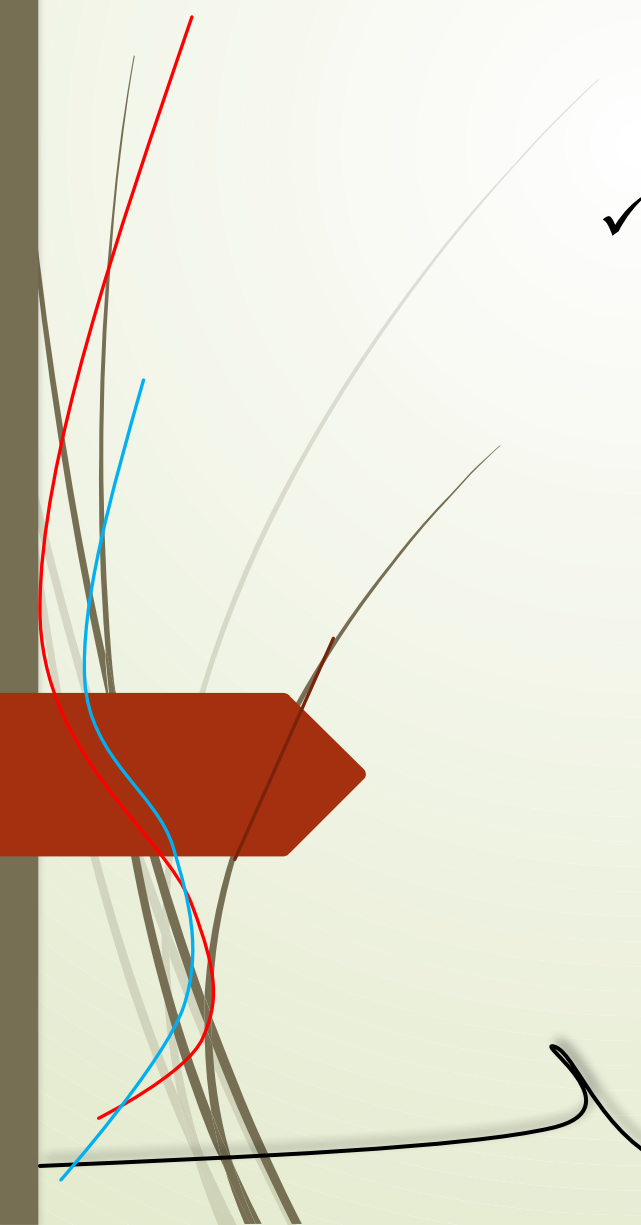
Algunas situaciones que requieren comunicación directa con el hardware:

- ❖ Software de equipos de prueba
- ❖ Sistemas operativos
- ❖ Software de conectividad de red (capa física)
- ❖ otras

Operadores a nivel de bit

- ✓ Se pueden realizar operaciones lógicas bit a bit en C
- ✓ Algunos operadores bit a bit son unarios y otros binarios
- ✓ Sintaxis general de los operadores bit a bit binarios:
operando1 operador operando2
- ✓ Sintaxis general de los operadores bit a bit unarios:
operador operando1
- ✓ Se utilizan en primer lugar para manipular los bits de un entero y en segundo lugar en sentencias if, if/else, while y do/while
- ✓ Los operandos de los operadores lógicos bit a bit deben ser enteros

Operadores binarios

- 
- ✓ Las operaciones que se puede realizar son:
 - ☐ Complementos (unario)
 - ☐ Desplazamientos a derecha (binario)
 - ☐ Desplazamiento a izquierda (binario)
 - ☐ Operación And (binario)
 - ☐ Operación OR inclusiva o simplemente OR (binario)
 - ☐ Operación OR-exclusiva o XOR binario).

Operadores bit a bit

✓ Tabla de operadores:

Operador	Nombre del operador
&	AND bit a bit
	OR inclusiva bit a bit
^	Operador XOR exclusiva bit a bit
~	Complemento a 1
>>	Desplazamiento hacia derecha
<<	Desplazamiento hacia izquierda

Operadores bit a bit

- ✓ Funcionalidad del operador and bit a bit

bit1	bit2	bit1 & bit2
0		0
0		0
0		0
1		1
1	0	
1		
1		

Operador And

CODIFICACIÓN

//Crear un algoritmo que permita testear si un determinado bit de un número está en 1, por ejemplo el bit posición 2^3 de info1 está seteado (estado lógico 1)

```
#include <stdio.h>
```

```
int main ( ) {
```

```
    char info1 = 24; // binario  00011000
```

```
    char test= 8; // 00001000
```

```
    printf ( " \n info1 = %d    test = %d ", info1 , test);
```

```
    if ( info1 & test ) printf ( " el bit de la posición  $2^3$  de info1 esta en 1");
```

```
    else printf ( " el bit de la posición  $2^3$  de info1 esta en 0");
```

```
    return 0, }
```


Operadores bit a bit

- ✓ Funcionalidad del operador **or** bit a bit (continuación)

bit1	bit2	bit1 bit2
0	0	0
0	1	1
1	0	1
1	1	1

Operador OR

Permite setear (poner a 1)
un bit determinado

Operadores bit a bit

- ✓ Funcionalidad del operador XOR bit a bit (continuación)

bit1	bit2	bit1 ^ bit2
0	0	0
0	1	1
1	0	1
1	1	0

Operador XOR

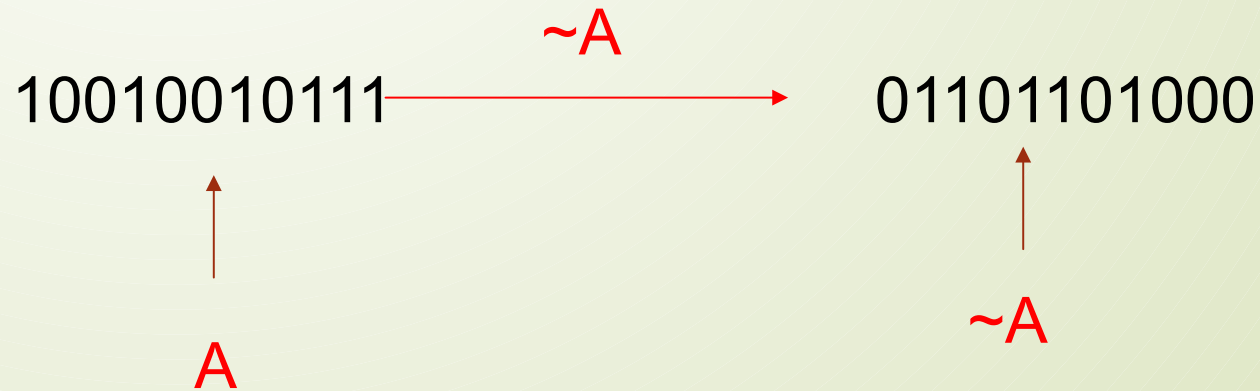
Operadores bit a bit

(continuación)

- ✓ Funcionalidad del operador complemento a 1 bit a bit
- ~ Complemento a 1

Descripción: Cambia los bit en estado 0 a 1, y viceversa

Ejemplo:



CODIFICACIÓN

//Aplicamos el complemento a uno a la representación binaria del número 17 en decimal

// 17 = 00010001_2 complemento a 1 11101110_2

```
#include <stdio.h>
```

```
int main ( ) {
```

```
    char info1 = 17; // binario 00010001
```

```
    char resul;
```

```
    printf ( "\n info1 = %d ", info1);
```

```
    resul = ~ info1;
```

```
    printf ( " result = %d", resul);
```

```
    return 0, }
```

Operadores bit a bit

Operador Desplazamiento a la izquierda

Descripción: desplaza los bits del primer operando1 hacia la izquierda según la cantidad de lugares especificado por el operando2, rellenando los lugares vacíos con 0 por derecha. El operando2 debe ser un número natural, es decir 1, 2, 3...

Ejemplo:

operando1: 00010000 operando2: 2

Después de la ejecución: $\text{operando1} \ll \text{operando2}$
operando1: 01000000

Operadores bit a bit

Operador desplazamiento a Derecha

Descripción: desplaza los bits del primer operando1 hacia la derecha según la cantidad de lugares especificado por el operando2, y el método de rellenado de lugares vacíos desde la izquierda es dependiente de la máquina.

Ejemplo:

operando1: 10010111 operando2: 2

Después de la ejecución: $\text{operando1} \gg \text{operando2}$
operando1: 00100101 (en este ejemplo los relleno con 0)

CODIFICACIÓN

Aplicamos corrimiento a derecha y a izquierda del numero 8 en 1

$$8 = 00001000_2$$

```
#include <stdio.h>
int main ( ) {
    char info1 = 8; // binario  00001000
    char desp= 1;
    char resul;
    printf ( "\n info1= %d << desp= %d ", info1, desp);
    resul = info1 << desp;
    printf ( " result = %d", resul);
    printf ( "\n info1= %d >> desp= %d ", info1, desp);
    resul = info1 >> desp;
    printf ( " result = %d", resul);
    return 0, }
```

Operadores bit a bit

Cada uno de los operadores a nivel de bit(a excepción del operador complemento a nivel de bit) tiene un operador de asignación correspondiente.

Operador de asignación a nivel de bit	
&=	Operador de asignación AND a nivel de bit
=	Operador de asignación OR inclusiva a nivel de bit
^=	Operador de asignación XOR exclusivo a nivel de bit
>>=	Operador de asignación de desplazamiento a la derecha
<<=	Operador de asignación de desplazamiento a la izquierda

Operadores a nivel de bit (continuación)

¿Precedencia de los operadores ?

¡Ahora trabaja usted!

Campos de bit

Agenda:

- ☐ Definición
- ☐ Declaración
- ☐ Ventajas y desventajas
- ☐ Usos típicos

Campos de bit

Los campos de bits son un número determinado de bits que pueden tener asociado o no un identificador.

Un campo de bit permite subdividir una estructura, unión o clase en un conjunto de bits en tamaño definido por el usuario. Para crear un campo de bit con un identificador opcional se realiza así:

```
<tipo_entero> Identificador_nombre : tamaño;
```

Los tipos asociados a los miembros que se deseen hacer campo de bit tiene que estar declarado a un tipo de dato entero o de carácter, como por ejemplo: char, unsigned char, short, unsigned short, long, unsigned long, int, unsigned int, __int64 o unsigned __int64.

Campos de bit

Continuación

Consideremos la siguiente construcción para campo de bit:

```
struct grupo_bit { unsigned a: 2;  
                  unsigned b: 3;  
                  unsigned c: 2;  
                  unsigned d: 1; };
```

- Esta definición tiene 4 campos de bits de tipo unsigned: a,b,c y d. Un campo de bit se declara colocando a continuación de un tipo int o unsigned el signo de dos puntos (:) seguido de una constante entera que representa el ancho del campo en bits.
- El ancho en bit es un entero comprendido entre 0 y el número total de bits que se utilizan para almacenar un int en la arquitectura en la que esté corriendo el programa (depende de la arquitectura del computador).
- En el ejemplo el miembro a está almacenado en 2 bit.

Campos de bit

Continuación

- Está permitido especificar un miembro sin identificador o nombre para el campo de bit, en tal situación este campo de bit se usa como relleno de la estructura. Ejemplo:

```
struct campo_bit { unsigned dato1: 4;  
                  unsigned dato2: 3;  
                  unsigned      : 5;  
                  unsigned dato3: 6;};
```

- La estructura `campo_bit` utiliza un campo de 5 bits de relleno sin identificador, el `dato3` (suponiendo una arquitectura de 2 byte para el tipo `unsigned`) se almacena en otra unidad de almacenamiento.

Características y limitaciones importantes

Ventajas y desventajas en la utilización de campo de bits:

- ✓ Los campos de bit permiten ahorrar almacenamiento aunque el proceso es más lento. Compromiso entre espacio de almacenamiento y tiempo de procesamiento.
- ✓ Permiten disponer variables de un bit, llamadas “semáforos”, para ello es conveniente usar los `#define`.
- ✓ Son dependientes de la arquitectura donde corre el programa incluso la organización en bits , byte y palabras puede variar dentro de las sucesivas versiones de un mismo compilador
- ✓ Los campos de bits no son un arreglo de bits, por lo tanto es un error tratar de acceder a los bits individuales de un campo de bits como si fueran elementos de un arreglo.

Características y limitaciones importantes

Ventajas y desventajas en la utilización de campo de bits (continuación):

- ✓ No está permitido aplicar el operador de referencia (&) con campo de bits porque no tienen direcciones.
- ✓ No se puede aplicar el operador sizeof.
- ✓ No puede ingresarse valores a un campo de bits mediante la función cin. Para poder leer valores desde el teclado se debe usar una variable entera auxiliar (ram dinámica) y luego copiar el contenido de esta variable auxiliar al campo de bit.

Buenas prácticas con campos de bits:

- Agrupar los campos de bits por tipo (juntos o próximos los del mismo tipo)
- Asegurarse de que están empaquetados dentro de sus áreas, ordenándolos de tal manera que ningún campo de bit tenga que saltar los límites de un área (según los tipos asociados a los miembros).
- Tratar de que la estructura esté tan rellena como sea posible.
- Forzar la alineación de media palabra (byte) mediante la directiva `#pragma option -a1`. Si fuese necesario conocer el tamaño de la estructura, se puede usar la directiva `#pragma sizeof` (identificador de la estructura), la cual devuelve el tamaño.

Usos y aplicaciones típicas

a) Una de las aplicaciones típicas es la posibilidad de disponer de variables de un bit para que funcionen como semáforos.

Usos y aplicaciones típicas

Ejemplo:

```
#define vacio 0x00
#define bit1 0x01 // 0000 0001
#define bit2 0x02 // 0000 0010
#define bit3 0x04 // 0000 0100
#define bit4 0x08 // 0000 1000
#define bit5 0x10 // 0001 0000
#define bit6 0x20 // 0010 0000
#define bit7 0x40 // 0100 0000
#define bit8 0x80 //1000 0000
if ( flags & bit1) { /*acciones*/} flags = bit2; // pone el bit 2 de flags en ON
flags = ~ bit3; // realiza un complemento a 1 -> bit 3 esta apagado ( OFF)
```


Usos y aplicaciones típicas

Continuación

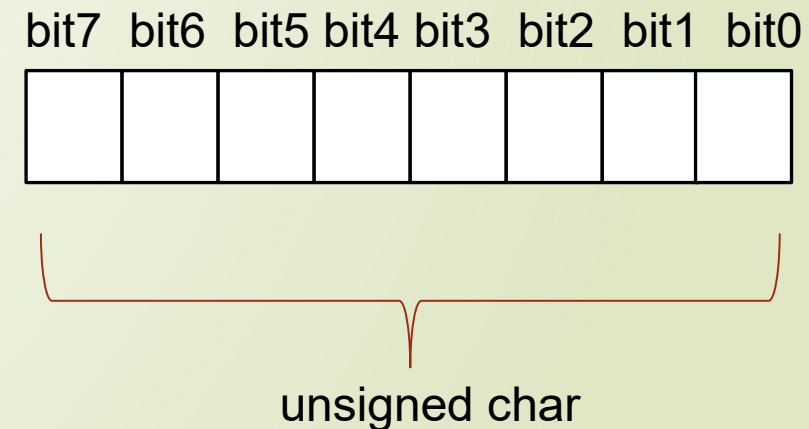
b) Cuando definimos campos de bits debemos usar siempre valores enteros sin signo porque el signo se almacena en el bit de mayor peso en la representación del número entero, esto puede falsear la representación de los datos almacenados en la estructura. Se presentan algunos casos.

Usos y aplicaciones típicas

Continuación

Caso1:

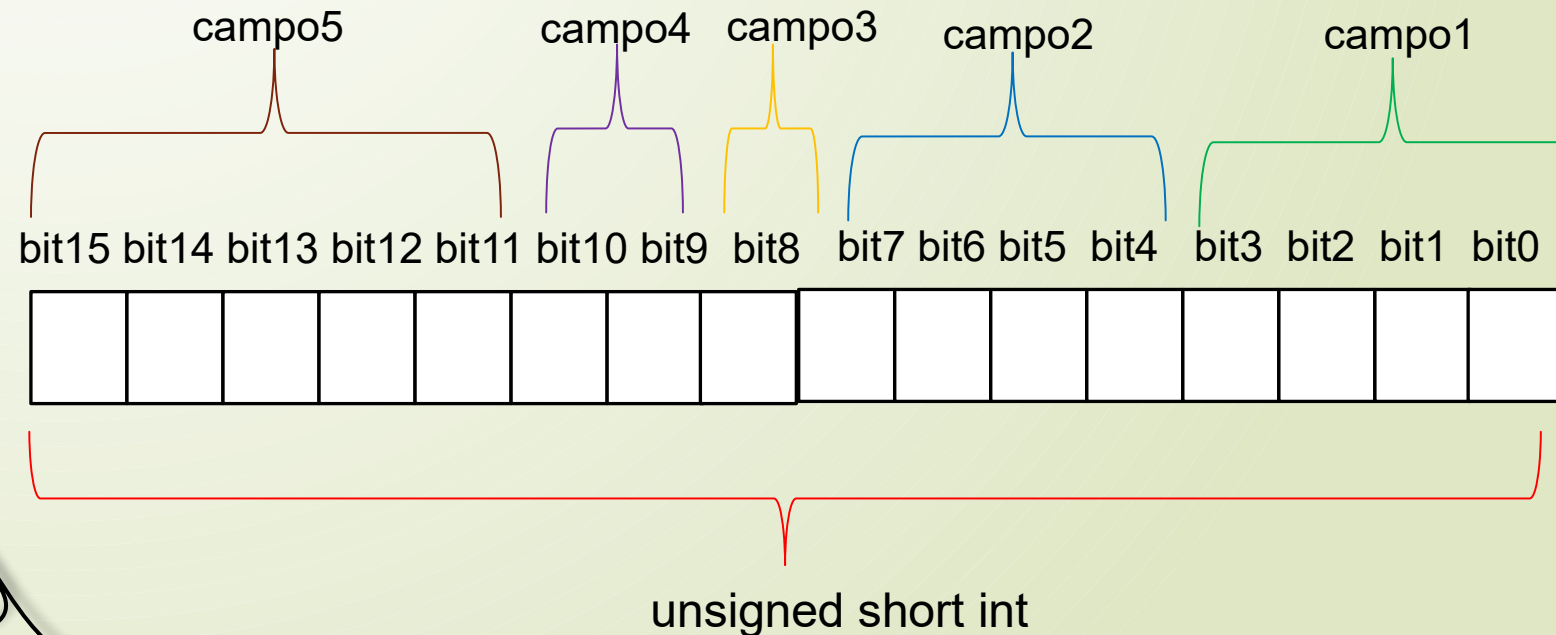
```
struct CampoBit {  
    unsigned char bit0:1;  
    unsigned char bit1:1;  
    unsigned char bit2:1;  
    unsigned char bit3:1;  
    unsigned char bit4:1;  
    unsigned char bit5:1;  
    unsigned char bit6:1;  
    unsigned char bit7:1;  
    unsigned char bit8:1; };
```



Usos y aplicaciones típicas

Continuación

Caso2: struct CampoBit2 { unsigned short int campo1:4;
unsigned short int campo2:4;
unsigned short int campo3:1;
unsigned short int campo4:2;
unsigned short int campo5:5;};

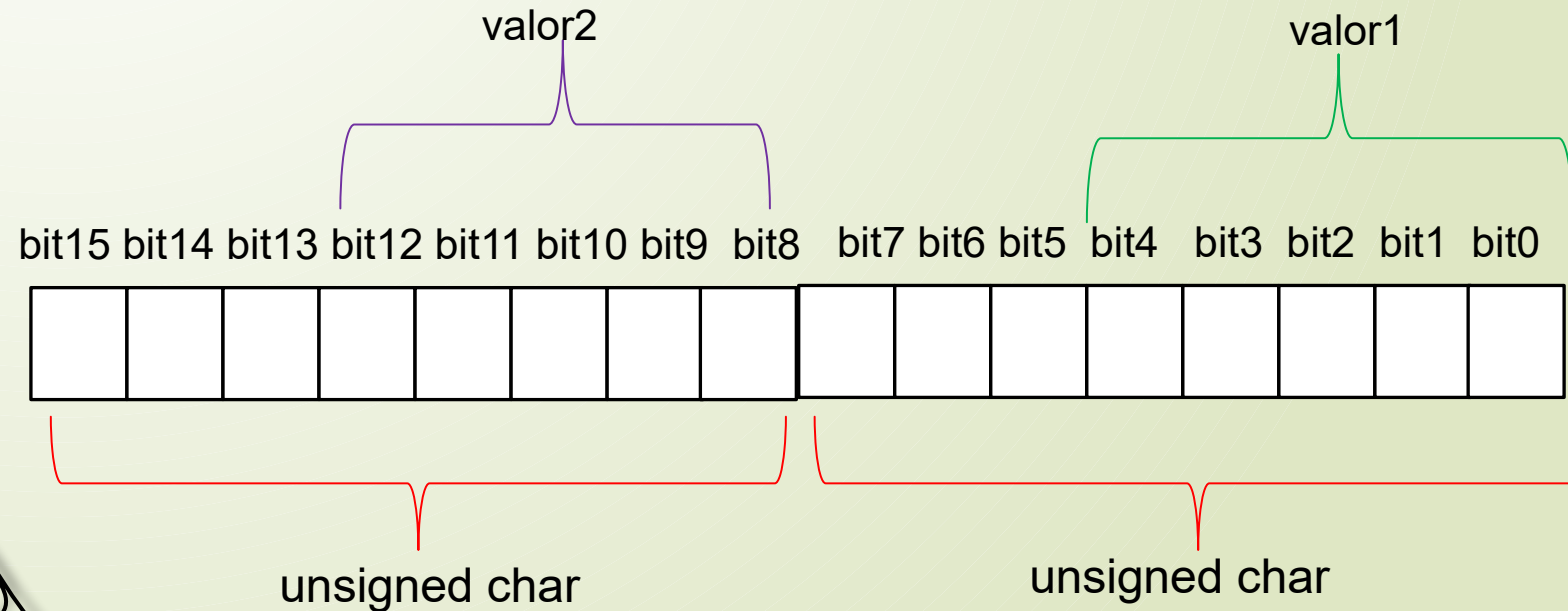


Usos y aplicaciones típicas

Continuación

Caso 3:

```
struct CampoBit3{ unsigned char valor1:5;  
                  unsigned char valor2:5;};
```



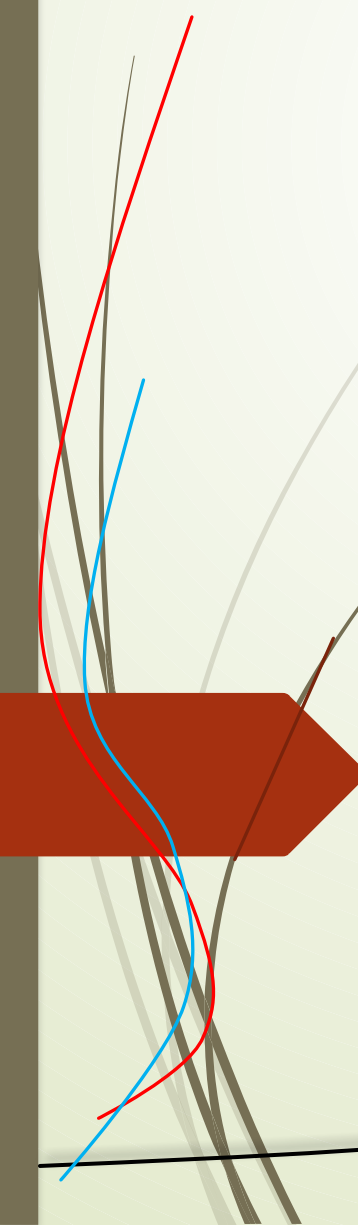
Usos y aplicaciones típicas

Continuación

c) También se pueden combinar miembros que sean campos de bit con otros que no lo sean dentro de la misma estructura. Una estructura que contiene campos de bit no debe ser exclusiva de este tipo de miembro, esta situación se muestra a continuación, en un ejemplo:

Usos y aplicaciones típicas

Continuación



```
#include < stdio>
using namespace std;
struct Colection { float dato1;
unsigned short int  campo1:4;
unsigned short int  campo2:4;
unsigned short int  campo3:1;
unsigned short int  campo4:2;
unsigned short int  campo5:5;
double  dato2};
int main ( )
{ struct Colection info;
info.campo1 = 10; //valores posibles [ 0   16]
info.campo3 =1; // valores posibles 0  1
printf ("\n %d", info.campo1); printf ("\n %d" ,info.campo3);
return 0;}
```


Ejercicios de aplicación

Ejercicio 1: Se desea implementar un programa que permita acceder a los 8 bit de un carácter en forma independiente (bit a bit) para mostrar por pantalla la representación de las letras A hasta la E, su número decimal asociado y su representación binaria.

```
#include <stdio>
using namespace std;
union { char n;
struct {unsigned A:1;
unsigned B:1;
unsigned C:1;
unsigned D:1;
unsigned E:1;
unsigned F:1;
unsigned G:1;
unsigned H:1;} info;
} var;
int main ( )
{ for ( var.n = 'A'; var.n <= ' E '; var.n ++ )
    printf ( "\n % c - %i- %u%u%u%u %u%u%u%u", var.n, var.n,
var.info.H, var.info.G, var.info.F, var.info.E, var.info.D, var.info.C,
var.info.B, var.info.A); return 0;}
```

SALIDA:

A - 65 – 0100 0001
B - 66 – 0100 0010
C - 67- 0100 0011
D - 68- 0100 0100
E - 69- 0100 0101

BIBLIOGRAFIA:

Obligatoria: capítulo 7 “Como programar en C/ C++”, Deitel y Deitel. 4ª edición, Prentice Hall.

Ampliatoria:

Miguel Angel Acera. (2015). *C/C++ curso de programación*. Editorial Anaya.

Martinez Fernandez R. (2014). *Programacion en C. Ejercicios*. Edición: Dextra Editorial S.L.



GRACIAS