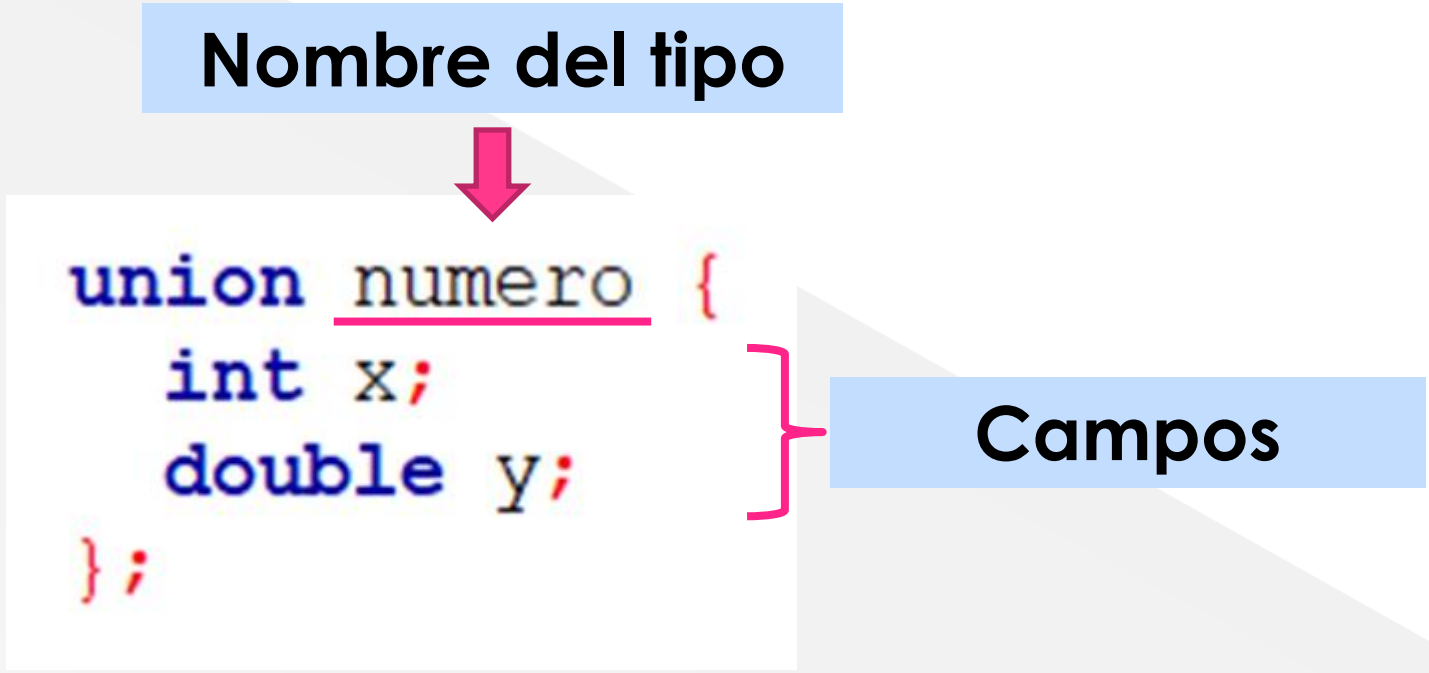
Tipo Union
Operadores a nivel de bits
Campos de bits
Constantes de enumeración

Tipo Union

- Al igual que una estructura, una unión también es un tipo de dato compuesto heterogéneo, pero con miembros que **comparten el mismo espacio de almacenamiento**.
- Sintaxis para declarar un tipo union

```
union Nom_Tipo {
    tipo_campo_1  nom_campo_1;
    tipo_campo_2  nom_campo_2;
     …
    tipo_campo_n  nom_campo_n;
};
```

Declaración de una Union



- Esta declaración no reserva memoria.
- Es sólo una declaración de tipo.

Declaración de variables

Es opcional pero si no existe, las variables sólo se declaran junto con la estructura

```
union numero {
   int x;
   double y;
} valor1;

union numero valor2;
```

Operaciones

- Las operaciones que pueden realizarse sobre una unión son:
  - Asignar una unión a otra unión del mismo tipo.
  - Obtener la dirección (&) de una variable unión.
  - Acceder a los campos de la unión.
- Las uniones, al igual que las estructuras, NO se pueden comparar usando los operadores == y !=

Inicializando Uniones en la declaración

- En una declaración, una variable *Union* puede ser inicializada con un valor del mismo tipo que el primer miembro de la unión.
- Ejemplo

```
union numero {
   int x;
   double y;
};

union numero valor = {10};
```

Esto es correcto porque x, el primer campo de la Union, es entero

Inicializando Uniones en la declaración

- En una declaración, una variable *Union* puede ser inicializada con un valor del mismo tipo que el primer miembro de la unión.
- Ejemplo

```
union numero {
   int x;
   double y;
};

union numero valor = {1.43};
```

El valor asignado será truncado porque x, el primer campo de la unión, es entero.

Union y struct

```c
#include <stdio.h>
union unionJob
{
    char name[32];
    float salary;
    int workerNo;
} uJob;

struct structJob
{
    char name[32];
    float salary;
    int workerNo;
} sJob;

int main()
{
    printf("Tamaño de la unión = %d bytes \n", sizeof(uJob));
    printf("Tamaño de la estructura = %d bytes", sizeof(sJob));
    return 0;
}
```

¿ Qué imprime?

Tamaño de la unión = 32 bytes
Tamaño de la estructura = 40 bytes

Ejemplo

¿ Qué imprime?

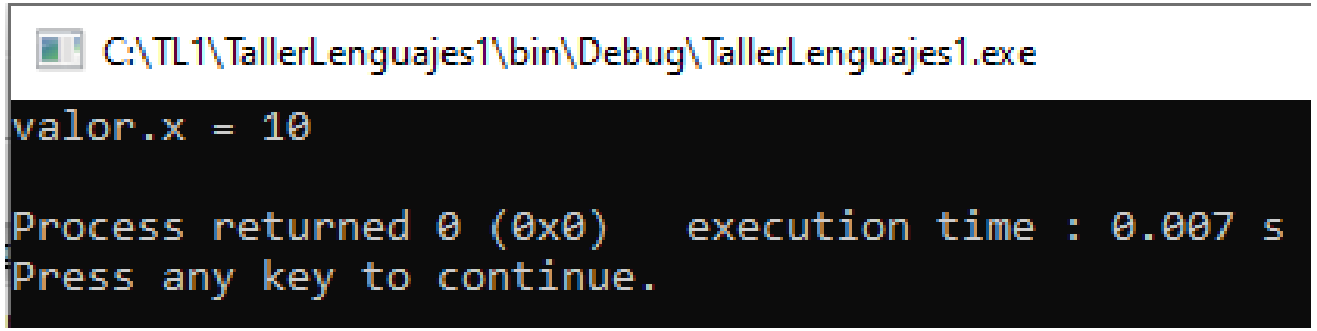
```c
#include <stdio.h>

union numero {
   int x;
   double y;
};

int main()
{ union numero valor = {10.78};

   printf("valor.x = %d \n", valor.x);

   return 0;
}
```



Ejemplo

¿ Qué imprime?

```c
union numero {
  int x;
  double y;
};
union numero valor;

valor.x = 10.34;
printf("Asigno 10.34 en el campo (int)\n");
printf("   (int) valor.x = %d\n", valor.x);
printf("(double) valor.y = %f\n\n", valor.y);

valor.y = 10.34;
printf("Asigno 10.34 en el campo (double)\n");
printf("   (int) valor.x = %d\n", valor.x);
printf("(double) valor.y = %f\n\n", valor.y);
```

```
Asigno 10.34 en el campo (int)
   (int) valor.x = 10
(double) valor.y = 0.000000
```

```
Asigno 10.34 en el campo (double)
   (int) valor.x = 2061584302
(double) valor.y = 10.340000
```

Ejemplo

- Utilice la siguiente estructura para operar con una fecha en dos formatos distintos

```
struct fecha {
    int tipo;
    union {
        time_t f_unix;
        char f_texto[11];
    } datos_fecha;
};
```

Indica cuál de los 2 formatos se está utilizando

Ejemplo

- Utilice la siguiente estructura para operar con una fecha en dos formatos distintos

```
struct fecha {
    int tipo;
    union {
        time_t f_unix;
        char f_texto[11];
    } datos_fecha;
};
```

Representa la fecha con un único número (cantidad de segundos desde el 01-01-1970)

Ejemplo

- Utilice la siguiente estructura para operar con una fecha en dos formatos distintos

```
struct fecha {
    int tipo;
    union {
        time_t f_unix;
        char f_texto[11];
    } datos_fecha;
};
```

Representa la fecha como una secuencia de caracteres. Ej: “10-04-2023”

```c
#include <stdio.h>
#include <string.h>
#include <time.h>

struct fecha {
    int tipo;
    union XX{
        time_t f_unix;
        char f_texto[11];
    } datos_fecha;
};

int main()
{   struct fecha F;

    F.tipo = 1;
    F.datos_fecha.f_unix = time(NULL);

    printf("Segundos desde el 01-01-1970 = %ld\n", F.datos_fecha.f_unix);

    F.tipo = 2;
    strcpy(F.datos_fecha.f_texto, "09-09-2024");
    printf("Fecha : %s\n", F.datos_fecha.f_texto);

    return 0;
}
```

¿Qué imprime?

Retorna la cantidad de segundos desde 01-01-1970

struct_Fecha_v1.c

```c
#include <stdio.h>
#include <string.h>
#include <time.h>

struct fecha {
    int tipo;
    union XX{
        time_t f_unix;
        char f_texto[11];
    } datos_fecha;
};

int main()
{   struct fecha F;

    F.tipo = 1;
    F.datos_fecha.f_unix = time(NULL);

    printf("Segundos desde el 01-01-1970 = %ld\n", F.datos_fecha.f_unix);

    F.tipo = 2;
    strcpy(F.datos_fecha.f_texto, "09-09-2024");
    printf("Fecha : %s\n", F.datos_fecha.f_texto);

    return 0;
}
```

¿Qué imprime?

Segundos desde el 01-01-1970 = 1725884319
Fecha : 09-09-2024

struct_Fecha_v1.c

```c
struct fecha {
   int tipo;
   union {
      time_t f_unix;
      char f_texto[11];
   } datos_fecha;
};
int main()
{  struct fecha F;

   F.tipo = 1;
   F.datos_fecha.f_unix = time(NULL);
   printf("Segundos desde el 01-01-1970 = %ld\n", F.datos_fecha.f_unix);

   struct tm *struct_tm = localtime(&F.datos_fecha.f_unix);

   printf ("Hoy es: %02d-%02d-%d\n", struct_tm->tm_mday,
                        1+struct_tm->tm_mon, 1900+struct_tm->tm_year);

   char fechaHora[50];

   strftime(fechaHora, sizeof(fechaHora), "%d-%m-%Y", struct_tm);

   printf("Fecha : %s\n", fechaHora);
   return 0;
}
```

struct_Fecha_v2.c

```c
struct fecha {
    int tipo;
    union {
        time_t f_unix;
        char f_texto[11];
    } datos_fecha;
};
int main()
{   struct fecha F;

    F.tipo = 1;
    F.datos_fecha.f_unix = time(NULL);
    printf("Segundos desde el 01-01-1970 = %ld\n", F.datos_fecha.f_unix);

    struct tm *struct_tm = localtime(&F.datos_fecha.f_unix);

    printf ("Hoy es: %02d-%02d-%d\n", struct_tm->tm_mday,
                          1+struct_tm->tm_mon, 1900+struct_tm->tm_year);

    char fechaHora[50];

    strftime(fechaHora, sizeof(fechaHora), "%d-%m-%Y", struct_tm);

    printf("Fecha : %s\n", fechaHora);
    return 0;
}
```

struct tm

Campo	Descripción
int tm_hour	hora (0 - 23)
int tm_isdst	Horario de verano enabled/disabled
int tm_mday	día del mes (1 - 31)
int tm_min	minutos (0 - 59)
int tm_mon	mes (0 - 11, 0 = Enero)
int tm_sec	segundos (0 - 60)
int tm_wday	día de la semana (0 - 6, 0 = domingo)
int tm_yday	día del año (0 - 365)
int tm_year	año desde 1900

struct_Fecha_v2.c

```c
struct fecha {
   int tipo;
   union {
      time_t f_unix;
      char f_texto[11];
   } datos_fecha;
};
int main()
{  struct fecha F;

   F.tipo = 1;
   F.datos_fecha.f_unix = time(NULL);
   printf("Segundos desde el 01-01-1970 = %ld\n", F.datos_fecha.f_unix);

   struct tm *struct_tm = localtime(&F.datos_fecha.f_unix);

   printf ("Hoy es: %02d-%02d-%d\n", struct_tm->tm_mday,
                        1+struct_tm->tm_mon, 1900+struct_tm->tm_year);

   char fechaHora[50];

   strftime(fechaHora, sizeof(fechaHora), "%d-%m-%Y", struct_tm);

   printf("Fecha : %s\n", fechaHora);
   return 0;
}
```

size_t strftime(char *,size_t,char *,struct tm *)

Formatea la información pasada mediante la estructura (struct tm) según el formato indicado en una cadena (char*) e imprime el resultado sobre otra cadena (char*) hasta un límite de caracteres (size_t).*

struct_Fecha_v2.c

```c
struct fecha {
    int tipo;
    union {
        time_t f_unix;
        char f_texto[11];
    } datos_fecha;
};
int main()
{   struct fecha F;

    F.tipo = 1;
    F.datos_fecha.f_unix = time(NULL);
    printf("Segundos desde el 01-01-1970 = %ld\n", F.datos_fecha.f_unix);

    struct tm *struct_tm = localtime(&F.datos_fecha.f_unix);

    printf ("Hoy es: %02d-%02d-%d\n", struct_tm->tm_mday,
                            1+struct_tm->tm_mon, 1900+struct_tm->tm_year);

    char fechaHora[50];

    strftime(fechaHora, sizeof(fechaHora), "%d-%m-%Y", struct_tm);

    printf("Fecha : %s\n", fechaHora);
    return 0;
}
```

```
Segundos desde el 01-01-1970 = 1725883862
Hoy es: 09-09-2024
Fecha : 09-09-2024
```

struct_Fecha_v2.c

Operadores a nivel de bits

- Las computadoras representan internamente todos los datos como secuencias de bits. Cada bit puede asumir un valor de 0 o 1. En la mayoría de los sistemas, una secuencia de 8 bits forma un byte.
- **Los operadores a nivel de bits se utilizan para manipular los bits de operandos enteros** (char, short, int y long; tanto signed como unsigned).
- Los enteros sin signo con frecuencia se utilizan con los operadores a nivel de bits.

	Operador	Descripción
&	AND a nivel de bits	Compara sus dos operandos bit a bit. Los bits en el resultado son setados a 1 si los bits correspondientes a **ambos** operandos valen 1.
|	OR a nivel de bits	Compara sus dos operandos bit a bit. Los bits en el resultado son setados a 1 si **al menos uno** de los bits correspondientes a los operandos valen 1.
^	XOR a nivel de bits	Compara sus dos operandos bit a bit. Los bits del resultado se establecen en 1, si **exactamente uno** de los bits correspondientes a los dos operandos es 1.
<<	Desplazamiento a la izquierda	Desplaza hacia la izquierda los bits del 1er.operando, el número de bits indicados por el 2do. operando; desde la derecha completa con bits en 0.
>>	Desplazamiento a la derecha	Desplaza hacia la derecha los bits del 1er.operando, el número de bits indicados por el 2do. operando; el método de llenado desde la izquierda depende de la máquina.
~	Complemento a uno	Todos los bits en 0 se cambian a 1 y viceversa.

Ejemplos

OpBits_Ejemplo1.c

```c
int a=1, b=2, c, d;

c = a & b;  //0001  &  0010 = 0000
d = 3 & b;  //0011  &  0010 = 0010

printf("1&2 = %d \n", c);
printf("3&2 = %d \n", d);
```

```
1 & 2 = 0
3 & 2 = 2
```

```c
int a=1, b=2, c, d;

c = a | b;  //0001  |  0010 = 0011
d = 3 | b;  //0011  |  0010 = 0011

printf("1|2 = %d \n", c);
printf("3|2 = %d \n", d);
```

```
1 | 2 = 3
3 | 2 = 3
```

Ejemplos

OpBits_Ejemplo1.c

```
int a=1, b=2, c, d;

c = a ^ b;  //0001  ^  0010 = 0011
d = 3 ^ b;  //0011  ^  0010 = 0001

printf("1^2 = %d \n", c);
printf("3^2 = %d \n", d);
```

```
1 ^ 2 = 3
3 ^ 2 = 1
```

```
int b=2;
unsigned char c;

c = ~b;  // 00000010 = 11111101
printf("~2 = %d \n", c);
```

```
~2 = 253
```

Ejemplos

OpBits_Ejemplo1.c

```c
int a, b;

a =  64;
b = a >> 3;  //01000000 --> 00001000
printf("64 >> 3 = %d \n", b);
```

```
64 >> 3 = 8
```

```c
int a, b;

a =  1;
b = a << 3;  //0001 --> 1000
printf("1 << 3 = %d \n", b);
```

```
1 << 3 = 8
```

Ejemplo

OpBits_esPar.c

```c
#include <stdio.h>
int main()
{
  unsigned x;

  printf("Ingrese un entero sin signo: ");
  scanf("%u", &x);

  printf("%u es %s \n", x, ( x % 2 ? "impar" : "par"));

  printf("%u es %s \n", x, ( x & 1 ? "impar" : "par"));

  return 0;
}
```

¿Qué pasa si usamos **&&** ?

Ejercicio

OpBits_verBits.c

- Lea un número entero sin signo e imprímalo utilizando su representación binaria.

```
 1  #include <stdio.h>
 2  void verBits( unsigned );
 3  int main()
 4  {
 5     unsigned x;
 6
 7     printf("Ingrese un entero sin signo: ");
 8     scanf("%u", &x);
 9
10     verBits(x);
11
12     return 0;
13  }
```

COMPLETAR

```
Ingrese un entero sin signo: 137
      137 - 00000000 00000000 00000000 10001001
```

EJEMPLO

```
El resultado de combinar los siguientes valores
     65535 = 00000000 00000000 11111111 11111111
         1 = 00000000 00000000 00000000 00000001
con el uso del operador de bits AND (&) es
         1 = 00000000 00000000 00000000 00000001

El resultado de combinar los siguientes valores
        15 = 00000000 00000000 00000000 00001111
       241 = 00000000 00000000 00000000 11110001
con el uso del operador de bits OR (|) es
       255 = 00000000 00000000 00000000 11111111

El resultado de combinar los siguientes valores
       139 = 00000000 00000000 00000000 10001011
       199 = 00000000 00000000 00000000 11000111
con el uso del operador de bits XOR (^) es
        76 = 00000000 00000000 00000000 01001100

El complemento a 1 de
     21845 = 00000000 00000000 01010101 01010101
es
4294945450 = 11111111 11111111 10101010 10101010
```

Salida del código C del archivo **OpBits_EjVarios.c**

EJEMPLO

```
El resultado del desplazamiento a izquierda de
       960 = 00000000 00000000 00000011 11000000
8 posiciones de bit con el uso del operador
de desplazamiento a izquierda << es
    245760 = 00000000 00000011 11000000 00000000


El resultado del desplazamiento a derecha de
       960 = 00000000 00000000 00000011 11000000
8 posiciones de bit con el uso del operador
de desplazamiento a derecha >> es
         3 = 00000000 00000000 00000000 00000011
```

Salida del código C del archivo **OpBits_Desplaza.c**

Desplazamientos de bits

- Si el operando derecho es negativo o si es mayor que el número de bits en el que el operando izquierdo está almacenado, el resultado del desplazamiento es indefinido.
- El desplazamiento a la derecha es dependiente de la implementación. Aplicar un desplazamiento a la derecha a un entero con signo puede ocasionar que los bits desocupados se llenen con ceros o que se llenen con unos.
- Los unos desplazados, si no pueden ser representados, se pierden.

Operadores de asignación a nivel de bits

Operador	Descripción
&=	Operador de asignación AND a nivel de bits
|=	Operador de asignación OR a nivel de bits
^=	Operador de asignación XOR a nivel de bits
<<=	Operador de asignación de desplazamiento a la izquierda
>>=	Operador de asignación de desplazamiento a la derecha

Campos de bits

- C permite a los programadores especificar el número de bits en el que un campo *unsigned* o *int* de una estructura o unión se almacena. A esto se le conoce como **campo de bits**.
- Los campos de bits permiten hacer un mejor uso de la memoria almacenando los datos en el número mínimo de bits necesario.
- Los miembros de un campo de bits deben declararse como *int* o *unsigned*.
- La manipulación de los campos de bits depende de la implementación.

CampoBit_Fecha.c

```c
#include <stdio.h>

struct datetime {
    unsigned int second : 6;
    unsigned int minute : 6;
    unsigned int hour : 5;
    unsigned int day : 5;
    unsigned int month : 4;
    unsigned int year : 6;
};

int main() {
    struct datetime dt = {30, 25, 11, 18, 9, 23};

    printf("sizeof(struct datetime) = %d bytes\n", sizeof(struct datetime));

    printf("Fecha y hora: %d/%d/%d %d:%d:%d\n",
           dt.day, dt.month, dt.year + 2000,
           dt.hour, dt.minute, dt.second * 2);

    return 0;
}
```

sizeof(struct datetime) = 4 bytes

Fecha y hora: 18/9/2023 11:25:60

CampoBit_LEDs.c

```c
#include <stdio.h>
struct LEDStatus {
    unsigned char LED1 : 1; // 1 bit para el LED1
    unsigned char LED2 : 1; // 1 bit para el LED2
    unsigned char LED3 : 1; // 1 bit para el LED3
    unsigned char LED4 : 1; // 1 bit para el LED4
    unsigned char : 4;      // relleno, 4 bits no utilizados
};

int main(){
    struct LEDStatus status;

    status.LED1 = 1;
    status.LED2 = 0;
    status.LED3 = 1;
    status.LED4 = 1;

    unsigned char byte = *(unsigned char*)&status;
    printf("Estado del LED: 0x%x\n", byte); // salida: Estado del LED: 0x1101

    status.LED2 = 1;
    byte = *(unsigned char*)&status;
    printf("Estado del LED: 0x%x\n", byte); // salida: Estado del LED: 0x1111

    return 0;
}
```

Ejemplo

- **Cara** almacena valores del 0 (As) al 12 (Rey); 4 bits pueden almacenar valores entre 0 y 15.

```
struct cartaBit {
    unsigned cara  : 4;
    unsigned palo  : 2;
    unsigned color : 1;
};
```

Cantidad de bits utilizados para almacenar el campo

Ejemplo

```
struct cartaBit {
    unsigned cara  : 4;
    unsigned palo  : 2;
    unsigned color : 1;
};
```

Cantidad de bits utilizados para almacenar el campo

- **Cara** almacena valores del 0 (As) al 12 (Rey); 4 bits pueden almacenar valores entre 0 y 15.
- **Palo** almacena valores del 0 al 3 (0 = Diamantes, 1 = Corazones, 2 = Tréboles, 3 = Espadas); 2 bits pueden almacenar valores entre 0 y 3.

Ejemplo

```
struct cartaBit {
    unsigned cara  : 4;
    unsigned palo  : 2;
    unsigned color : 1;
};
```

Cantidad de bits utilizados para almacenar el campo

- **Cara** almacena valores del 0 (As) al 12 (Rey); 4 bits pueden almacenar valores entre 0 y 15.
- **Palo** almacena valores del 0 al 3 (0 = Diamantes, 1 = Corazones, 2 = Tréboles, 3 = Espadas); 2 bits pueden almacenar valores entre 0 y 3.
- **Color** almacena 0 (Rojo) o 1 (Negro); 1 bit puede almacenar 0 o 1.

Ejemplo

CampoBit_EjMazo.c

- Utilice una estructura formada por campos de bits para almacenar un mazo de 52 cartas de póker.
  - Defina la estructura
  - Cargue las cartas en el mazo
  - Muestre el mazo en pantalla

EJEMPLO

```
Carta     Palo    Color        Carta     Palo    Color
Carta:  0 Palo: 0 Color: 0     Carta:  0 Palo: 2 Color: 1
Carta:  1 Palo: 0 Color: 0     Carta:  1 Palo: 2 Color: 1
Carta:  2 Palo: 0 Color: 0     Carta:  2 Palo: 2 Color: 1
Carta:  3 Palo: 0 Color: 0     Carta:  3 Palo: 2 Color: 1
Carta:  4 Palo: 0 Color: 0     Carta:  4 Palo: 2 Color: 1
Carta:  5 Palo: 0 Color: 0     Carta:  5 Palo: 2 Color: 1
Carta:  6 Palo: 0 Color: 0     Carta:  6 Palo: 2 Color: 1
Carta:  7 Palo: 0 Color: 0     Carta:  7 Palo: 2 Color: 1
Carta:  8 Palo: 0 Color: 0     Carta:  8 Palo: 2 Color: 1
Carta:  9 Palo: 0 Color: 0     Carta:  9 Palo: 2 Color: 1
Carta: 10 Palo: 0 Color: 0     Carta: 10 Palo: 2 Color: 1
Carta: 11 Palo: 0 Color: 0     Carta: 11 Palo: 2 Color: 1
Carta: 12 Palo: 0 Color: 0     Carta: 12 Palo: 2 Color: 1
Carta:  0 Palo: 1 Color: 0     Carta:  0 Palo: 3 Color: 1
Carta:  1 Palo: 1 Color: 0     Carta:  1 Palo: 3 Color: 1
Carta:  2 Palo: 1 Color: 0     Carta:  2 Palo: 3 Color: 1
Carta:  3 Palo: 1 Color: 0     Carta:  3 Palo: 3 Color: 1
Carta:  4 Palo: 1 Color: 0     Carta:  4 Palo: 3 Color: 1
Carta:  5 Palo: 1 Color: 0     Carta:  5 Palo: 3 Color: 1
Carta:  6 Palo: 1 Color: 0     Carta:  6 Palo: 3 Color: 1
Carta:  7 Palo: 1 Color: 0     Carta:  7 Palo: 3 Color: 1
Carta:  8 Palo: 1 Color: 0     Carta:  8 Palo: 3 Color: 1
Carta:  9 Palo: 1 Color: 0     Carta:  9 Palo: 3 Color: 1
Carta: 10 Palo: 1 Color: 0     Carta: 10 Palo: 3 Color: 1
Carta: 11 Palo: 1 Color: 0     Carta: 11 Palo: 3 Color: 1
Carta: 12 Palo: 1 Color: 0     Carta: 12 Palo: 3 Color: 1
```

Salida del código C del archivo **CampoBit_EjMazo.c**

Campos de Bits

- Los campos de bits ayudan a reducir la cantidad de memoria que necesita el programa.
- Aunque los campos de bits ahorran espacio, utilizarlos puede ocasionar que el compilador genere código en lenguaje máquina de ejecución lenta.
  - *Esto ocurre debido a que éste toma operaciones adicionales en lenguaje máquina para acceder sólo a porciones de una unidad de almacenamiento direccionable.*

Compromiso entre el espacio de memoria y el tiempo de ejecución

Campos de bits. Errores frecuentes

- Intentar acceder a bits individuales de un campo de bits, como si fueran elementos de un arreglo, es un error de sintaxis. Los campos de bits no son “arreglos de bits”.
- Intentar tomar la dirección de un campo de bits (el operador & no debe utilizarse con campos de bits, ya que éstos no tienen direcciones).

Constantes de enumeración

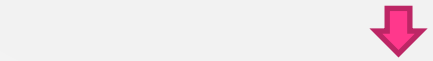
- Una enumeración es un conjunto de constantes de enumeración enteras representadas por identificadores.
- Los valores de una enumeración empiezan por 0 a menos que se especifique lo contrario, y se incrementan en 1.
- Ejemplo

```
enum meses { ENE, FEB, MAR, ABR, MAY, JUN,
             JUL, AGO, SEP, OCT, NOV, DIC };
```

Crea un nuevo **tipo**, enum meses, en el que los identificadores se corresponden con los valores enteros de 0 a 11

Constantes de enumeración

- Una enumeración es un conjunto de constantes de enumeración enteras representadas por identificadores.
- Los valores de una enumeración empiezan por 0 a menos que se especifique lo contrario, y se incrementan en 1.
- Ejemplo



```
enum meses { ENE = 1, FEB, MAR, ABR, MAY, JUN,
             JUL, AGO, SEP, OCT, NOV, DIC };
```

Crea un nuevo **tipo**, enum meses, en el que los identificadores se corresponden con los valores enteros de 1 a 12

Enumeración. Tipo y tamaño

- En ANSI C, las expresiones que definen el valor de una constante de enumerador siempre tienen el **tipo int**.
- El almacenamiento asociado a una variable de enumeración es el necesario para un único valor int.
- Una constante de enumeración o un valor de tipo enumerado se pueden usar en cualquier lugar donde el lenguaje C permita una expresión de tipo entero.

enum_variableMeses.c

Declaración de variable de enumeración

```c
#include <stdio.h>
enum meses { ENE , FEB, MAR, ABR, MAY, JUN,
             JUL, AGO, SEP, OCT, NOV, DIC };
int main()
{
    enum meses mes;

    mes = MAR;

    printf("La variable mes vale %d\n", mes);

    return 0;
}
```

Una variable del tipo Enumeración almacena uno de los valores del conjunto de enumeración definido por ese tipo

```
La variable mes vale 2
```

Ejemplo

enum_Meses.c

```c
#include <stdio.h>
#include <stdlib.h>
enum meses { ENE = 1, FEB, MAR, ABR, MAY, JUN,
                  JUL, AGO, SEP, OCT, NOV, DIC };
int main()
{
    enum meses mes;
    const char *nombreMes[] = { "", "Enero", "Febrero",
        "Marzo", "Abril", "Mayo", "Junio", "Julio",
        "Agosto", "Septiembre", "Octubre","Noviembre",
        "Diciembre" };

    /* ciclo a través de los meses */
    for ( mes = ENE; mes <= DIC; mes++ )
       printf( "%2d%11s \n", mes, nombreMes[ mes ] );

    return 0;
}
```

```
 1      Enero
 2    Febrero
 3      Marzo
 4      Abril
 5       Mayo
 6      Junio
 7      Julio
 8     Agosto
 9 Septiembre
10    Octubre
11  Noviembre
12  Diciembre
```

Identificadores y valores

- Los identificadores de una enumeración deben ser únicos (incluye también nombres de variables).
- El valor de cada constante de enumeración puede establecerse explícitamente en la definición, asignándole un valor al identificador.
- Varios miembros de una enumeración pueden tener el mismo valor constante.

Enumeración con valores repetidos

```c
#include <stdio.h>
enum Dupes
{   Base, /* Vale 0 */
    One,  /* Vale Base + 1 */
    Two,  /* Vale One + 1 */
    Negative = -1,
    AnotherZero /* Vale Negative + 1 == 0 */
};

int main()
{
  printf("Base = %d\n", Base);
  printf("Uno = %d\n", One);
  printf("Dos = %d\n", Two);
  printf("Negativo = %d\n", Negative);
  printf("otroCero = %d\n", AnotherZero);

  return 0;
}
```

```
Base = 0
Uno = 1
Dos = 2
Negativo = -1
otroCero = 0
```

enum_Repetidos.c

enum_Boolean.c

Enum vs #define

- Las enumeraciones proporcionan una alternativa a la directiva de preprocesador *#define* con las ventajas de que los valores se pueden generar automáticamente.

```
/* declara el tipo BOOLEAN */
enum BOOLEAN
{
    false,    /* false = 0, true = 1 */
    true
};

/* dos variables de tipo BOOLEAN */
enum BOOLEAN end_flag, match_flag;
```

Enum vs #define

- Las enumeraciones siguen las reglas de alcance.
- A las variables *enum* se les asignan valores automáticamente.

```
#define Working 0
#define Failed 1
#define Freezed 2
```

```
enum state {Working,
            Failed,
            Freezed};
```

Constante de enumeración anónima

- Los tipos de enumeración también se pueden declarar sin darles un nombre:

```
enum { buffersize = 256 };
static unsigned char buffer[buffersize] = {0};

printf("buffer ocupa %d bytes\n", sizeof(buffer));
```

- Esto permite definir **constantes en tiempo de compilación** de tipo *int* que, como en este ejemplo, se pueden usar para indicar la cantidad de elementos del vector.

Indique qué imprime

```
int a=5, b=6;
printf("a & b = %d", a & b);
```

```
int c=2, d=7;
printf("c && d = %d", c && d);
```

Indique qué imprime

```
enum {UNO, DOS, TRES=0, CUATRO} p;
int suma=0, V[]={1,2,3,4,5,6,7,8,9};

for (p=UNO; p<CUATRO; p++)
    suma = suma + V[p];

printf("suma = %d", suma);
```

Indique qué imprime

```
enum {UNO, DOS, TRES=0, CUATRO} p;
int suma=0, V[]={1,2,3,4,5,6,7,8,9};

for (p=UNO; p<CUATRO; p++)
    suma = suma + V[p];

printf("suma = %d", suma);
```

Si sacamos “=0”?

Indique los errores

```c
struct persona {
   char nom[15];
   int edad;
}

struct empresa {
   char nom[15];
   char direccion[30];
}
```

```c
struct persona p;
struct empresa e;

printf("Ingrese el nombre : ");
scanf("%s", e.nom);

p.nom = e.nom;

printf("Ingrese la edad :");
scanf("%d", &p.edad);

printf("nombre : %s  - edad : %d",
        p.nom, p.edad);
```

Ejercicio: ¿Cuánto vale X?

```
/* determina si num es un múltiplo de X */
int multiplo( int num )
{
    int i;
    int mascara = 1;
    int mult = 1;

    for ( i = 1; i <= 10; i++, mascara <<= 1 ) {
          if ( ( num & mascara ) != 0 ) {
                mult = 0;
                break;
           }
     }
     return mult;
}
```

Ejer_FuncionMultiplo.c

Ejercicio: ¿Qué retorna la función?

```
int misterio( unsigned bits )
{
    unsigned i;
    unsigned mascara = 1 << 31;
    unsigned total = 0;

    for ( i = 1; i <= 32; i++, bits <<= 1 )

        if ( ( bits & mascara ) == mascara )
            total++;

    return !( total % 2 ) ? 1 : 0;
}
```

Ejer_FuncionMisterio.c

Ejercicio 1

- Desarrolle un programa que utilice un tipo de datos adecuado para modelar figuras geométricas bidimensionales y sus propiedades: circulo (radio), cuadrado (lado) y rectángulo (2 lados).
  - Defina los tipos de datos necesarios para implementar el tipo de datos figura
  - Implemente una función que reciba un arreglo de figuras y una cantidad de figuras e imprima para cada figura, el tipo de figura con sus propiedades y el perímetro correspondiente.

Ejercicio 2

- Implemente un tipo de datos Fecha para almacenar día, mes y año teniendo en cuenta las siguientes observaciones:
- Como las comparaciones entre fechas son algo engorrosas, utilice una unión para trabajar una misma fecha con dos representaciones alternativas:
  - los tres campos que componen la fecha por separados
  - una representación alternativa que permita compararlas directamente (estudiar orden y tamaño de cada campo de la fecha).
- Implemente un programa que compare distintas fechas para demostrar que esta estrategia funciona (puede aprovechar la declaración de las variables para asignar las fechas).

```
union TipoFecha{
   struct {
        char dia;
        char mes;
        short int anio;
     } TFecha;
   unsigned int FechaNro;
};
```