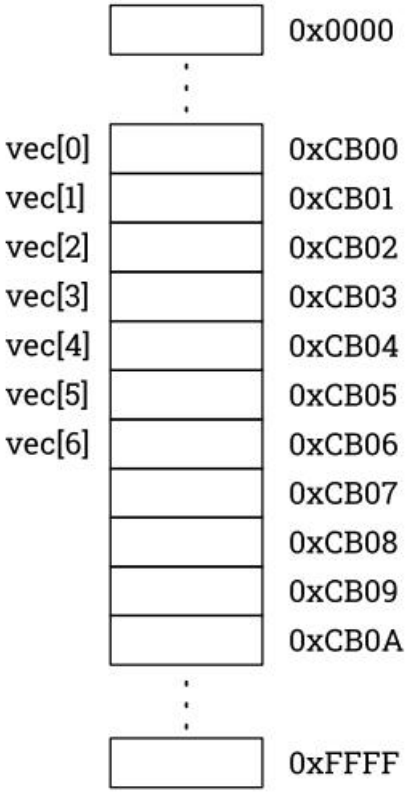
ARREGLOS EN C

Son un conjunto de posiciones de memoria contiguos, para almacenar valores del mismo tipo. Su identificador (nombre) debe respetar las normas generales.

Se puede acceder a determinado valor mediante su nombre e índice de la posición del mismo.




```
vec[3] = 15;
```

El índice puede ser una variable o cualquier expresión

```
for (int i = 0; i < 7; i++)
printf ("%d\n", vec[i]);
```

El primer elemento siempre es de índice [0].

Definición:

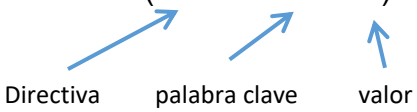
Tipo identificador[tamaño];

```
int vec[100];
```

El verdadero tamaño máx. depende de varios factores relacionados a la memoria del programa. La manera más cómoda de determinar el tamaño del arreglo es con la directiva de preprocesador *#define.*

#define:

Se utiliza para definir constantes simbólicas (#define TAM 10)



Antes de la compilación, el preprocesador reemplaza las ctes. simbólicas con su valor correspondiente.

No llevan punto y coma (;) en su declaración.

Inicialización de arreglos:

Se inicializan de manera parecida a una variable, sólo que en este caso, al poder guardar más de 1 solo valor, deberemos colocar tales valores entre llaves y separados por comas, los elementos entre llaves se guardarán en orden desde el índice 0:

```
int vec[5] = {3, 5, 2, 10, 4};
```

Tamaño

i[0] i[1]

vec[0]	3
vec[1]	5
vec[2]	2
vec[3]	10
vec[4]	4

Si los valores inicializados son menos que el tamaño del mismo, el resto de elementos se inicializarán en 0:

```
int vec[5] = {3, 5, 2,};
```

vec[0]	3
vec[1]	5
vec[2]	2
vec[3]	0
vec[4]	0

NO se pueden inicializar más elementos que el tamaño del arreglo. El compilador da error.

Se puede inicializar todo el arreglo poniendo un solo 0 entre llaves:

```
int vec[5] = {0};
```

(técnicamente se inicializa el primer elemento en 0 y el resto igual por lo explicado anteriormente)

Se puede omitir el tamaño entre llaves, siempre que se usen llaves para inicializar:

```
int vec[] = {3, 5, 2, 10, 4};
```

(el tamaño será la cant. de elementos inicializados)

No se pueden omitir el tamaño y las llaves al mismo tiempo, esto da error.

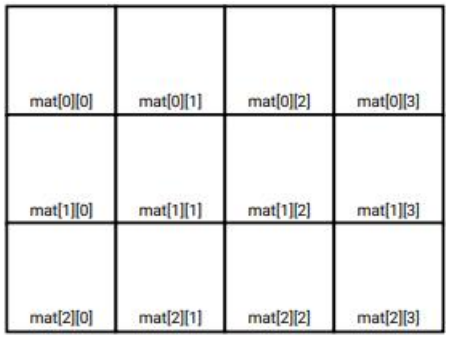
```
int vec[];
```

← ✕ MAL

ARREGLOS DE 2 DIMENSIONES:

```
int mat[3][4];
```

Cant. filas — Cant. columnas

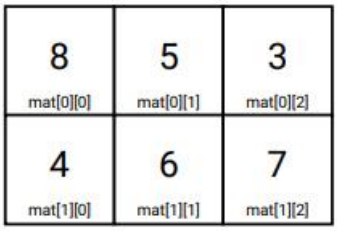


Para acceder a sus elementos hay que utilizar los índices de filas y columnas.

Para inicializarlos:

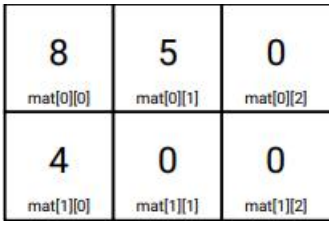
```
int mat[2][3] = { {8,5,3}, {4,6,7} };
```

(llaves para las filas y llaves para las columnas)



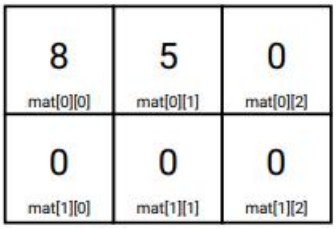
Se pueden incializar de manera incompleta de tal forma que el resto de elementos queden en 0:

```
int mat[2][3] = { {8,5}, {4} };
```



Pueden faltar filas también:

```
int mat[2][3] = { {8,5} };
```



O se pueden inicializar todos los elementos en 0:

`int mat[2][3] = { {0} };` O `int mat[2][3] = {0};`

Si se utiliza otro número da *warning.* Aunque se vaya a inicializar el primer elemento con el n° especificado de todas maneras, no es lo recomendable.

Acceso a sus elementos:

```
int mat[3][3] = {0};

mat[1][1] = 1;

for (int i = 0; i < 3; i++ ) {
  for (int j = 0; j < 3; j++ )
    printf("%d ", mat[i][j]);
  printf("\n");
}
```

```
0 0 0
0 1 0
0 0 0
```

Para cargar el arreglo:

```
#include <stdio.h>

#define N 3

int main (void) {
  int i,j;
  int mat[3][3] = {0};

  for (int i = 0; i < N; i++ )
    for (int j = 0; j < N; j++ ) {
      printf("Ingrese el elemento (%d,%d): ", i, j);
      scanf("%d", &mat[i][j]);
    }

  for ( i = 0; i < N; i++ ) {
    for ( j = 0; j < N; j++ )
      printf("%d ", mat[i][j]);
    printf("\n");
  }
  return 0;
}
```

PASAJE DE ARREGLOS A FUNCIONES

Los cambios realizados sobre el arreglo en una función **afectan al arreglo original**. En el llamado no deben usarse los corchetes, solo el nombre. En la definición del arreglo en la lista de parámetros SI debe tener corchetes.

```
#include <stdio.h>

#define N 10

void carga (int vec[N])
{
  for (int i = 0; i < N; i++)
    vec[i] = i;
}

int main (void)
{
  int vec[N] = {0};

  carga(vec);
  for (int i = 0; i < N; i++)
    printf("%d ", vec[i]);

  return 0;
}
```

```
0 1 2 3 4 5 6 7 8 9
```

También se pueden dejar los corchetes vacíos.

```
void carga (int vec[])
{
  for (int i = 0; i < N; i++)
    vec[i] = i;
}
```

Una buena práctica es no incluir valores globales en la función, sino más bien,declarar una variable que contenga el tamaño del arreglo. Por lo tanto se dejarían los corchetes del arreglo vacíos.

```
void carga (int vec[], int n)
{
  for (int i = 0; i < n; i++)
    vec[i] = i;
}
```

En caso de un arreglo de más dimensiones, el procedimiento es el mismo solo que puede omitirse solo el primer tamaño del arreglo.

```c
void carga (int mat[][M], int n, int m)
{
  int c = 0;

  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
      mat[i][j] = c++;

}
```

ALGORITMOS

Intercambio:

Para intercambiar elementos en un arreglo se debe usar una variable temporal para hacer un “backup” de 1 de los valores:

```c
#include <stdio.h>
// u6-swap-1.c

int main (void) {
  int arreglo[5] = {90,10,20,30,0};
  int tmp;

  tmp = arreglo[0];
  arreglo[0] = arreglo[4];
  arreglo[4] = tmp;

  for (int i = 0; i < 5; i++)
    printf("%d\n", arreglo[i]);

  return 0;
}
```

Luego de salvado el valor, se lo puede pisar con el nuevo elemento. Y finalmente se recupera el valor salvado en la variable y se lo coloca en el lugar del valor anterior.

Tipos de ordenamiento

Búsqueda lineal/secuencial: Sirve para encontrar un valor en el arreglo.

```c
#include <stdio.h>
// u6-max-1.c

int main (void) {
  int arreglo[5] = {4,1,6,5,3};
  int max;

  max = arreglo[0];
  for (int i = 1; i < 5; i++)
    if (arreglo[i] > max)
      max = arreglo[i];

  printf("máximo = %d\n", max);

  return 0;
}
```

1) Inicializar “max” con el 1er elemento del arreglo.
2) Recorrer el arreglo buscando un valor mayor a “max”.
3) Si se lo encuentra se actualiza “max”,

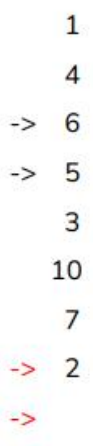
Método de la burbuja: Compara cada elemento con el siguiente y lo cambia en base al criterio de ordenamiento.

$$\texttt{int vec[8]} = \{4, 1, 6, 5, 3, 10, 7, 2\};$$

Ordenemos este arreglo de menor a mayor.
1) Se compara el elemento 0 con el 1, si el 0 es mayor que el 1, se intercambian de la siguiente manera:

```
for (int i = 0; i < N-1; i++) {
    if (vec[i] > vec[i+1]) {
        int tmp = vec[i];
        vec[i] = vec[i+1];
        vec[i+1] = tmp;
    }
}
```

2) En el for se utiliza N-1 ya que cuando llega al final:



Intenta comparar el último elemento con memoria vacía o basura. Esto es un error lógico.

Como se ve en la imagen, el for terminó su recorrido, sin embargo, el arreglo no está ordenado. Para esto hay q agregar un segundo for que lo englobe:

```
for (int j = 0; j < N-1; j++) {
    for (int i = 0; i < N-1; i++) {
        if (vec[i] > vec[i+1]) {
            int tmp = vec[i];
            vec[i] = vec[i+1];
            vec[i+1] = tmp;
        }
    }
}
```

Se puede ver que en la primera pasada el elemento mayor del arreglo queda en su posición correspondiente siempre, en la segunda pasada la última pregunta no hace falta porque nunca será verdad, y así sucesivamente. Por lo que este algoritmo se puede **mejorar** para evitar que se hagan preguntas demás.

Burbuja mejorada:

```
for (int j = 0; j < N-1; j++) {
    for (int i = 0; i < N-1-j; i++) {
        if (vec[i] > vec[i+1]) {
            int tmp = vec[i];
            vec[i] = vec[i+1];
            vec[i+1] = tmp;
        }
    }
}
```

Método de inserción: Consiste en ordenar cada elemento respecto a los elementos previamente ordenados.

Supongamos un conj. Donde todos los elementos a la izq. del elemento rojo están ordenados de menor a mayor

1 4 5 6 3 10 7 2

Buscaremos la posición del elemento rojo en la lista ordenada:

1 3 4 5 6 10 7 2

Ahora:

1 3 4 5 6 10 7 2

Se supone que el conj. a la izq ya está ordenado, solo es necesario preguntar por el elemento más a la derecha del conjunto.

1 3 4 5 6 10 7 2

Si no es mayor el rojo se mueve a la derecha y se vuelve a empezar.

```
void ordenar_menor_a_mayor_insercion (int vec[], int n) {
  for(int j = 1; j < n; j++) {
    int tmp = vec[j];
    int i = j;
    while(i > 0 && vec[i-1] > tmp ) {
      vec[i] = vec[i-1];
      i--;
    }

    vec[i] = tmp;
  }
}
```

Ordenamiento rápido(Quicksort):

Toma el primer elemento del arreglo y lo coloca en su lugar final, donde todos los elementos a su izq sean menores y lo de su derecha mayores. Ambos grupos están desordenados, pero en cada uno se hace lo mismo. Esto se repite con cada subconjunto que se va generando. Esto se repite hasta que cada elemento esté en su lugar.

1) Tomamos el 1er elemento de la izq.:

37 2 6 4 89 8 10 12 68 45

2) Buscamos desde la derecha el primer elemento que sea menor que el seleccionado:

37 2 6 4 89 8 10 12 68 45

3) Los intercambiamos:

12 2 6 4 89 8 10 37 68 45

4) Desde la izq pero comenzando con el siguiente elemento al recién colocado, se busca el primer elemento mayor al seleccionado:

12 2 6 4 89 8 10 37 68 45

5) Intercambiamos:

12 2 6 4 37 8 10 89 68 45

6) Desde la derecha comenzando desde el sig. al elemento recién colocado, buscamos el primer elemento menor al seleccionado:

12 2 6 4 37 8 10 89 68 45

7) Intercambiamos:

12 2 6 4 10 8 37 89 68 45

8) Ahora a la derecha ya no hay elementos menores al seleccionado, y a la izq no hay elementos mayores.

9) Sin embargo estos elementos no están ordenados, por lo que hay que repetir el proceso, el cual finalizará cuando el elemento seleccionado sea el único subconjunto.

```c
void quicksort (int item[], int left, int right)
  int i, j, temp;

  i = left; j = right;

  do {
    while (item[i] < item[j] && i < j) j--;

    if (i < j) {
      temp = item[i];
      item[i] = item[j];
      item[j] = temp;
      i++;
    }

    while ( item[i] < item[j] && i < j) i++;

    if(i < j) {
      temp = item[i];
      item[i] = item[j];
      item[j] = temp;
      j--;
    }
  } while (i < j);

  if (left < j) quicksort(item, left, j-1);
  if (i < right) quicksort (item, i+1, right);
}
```

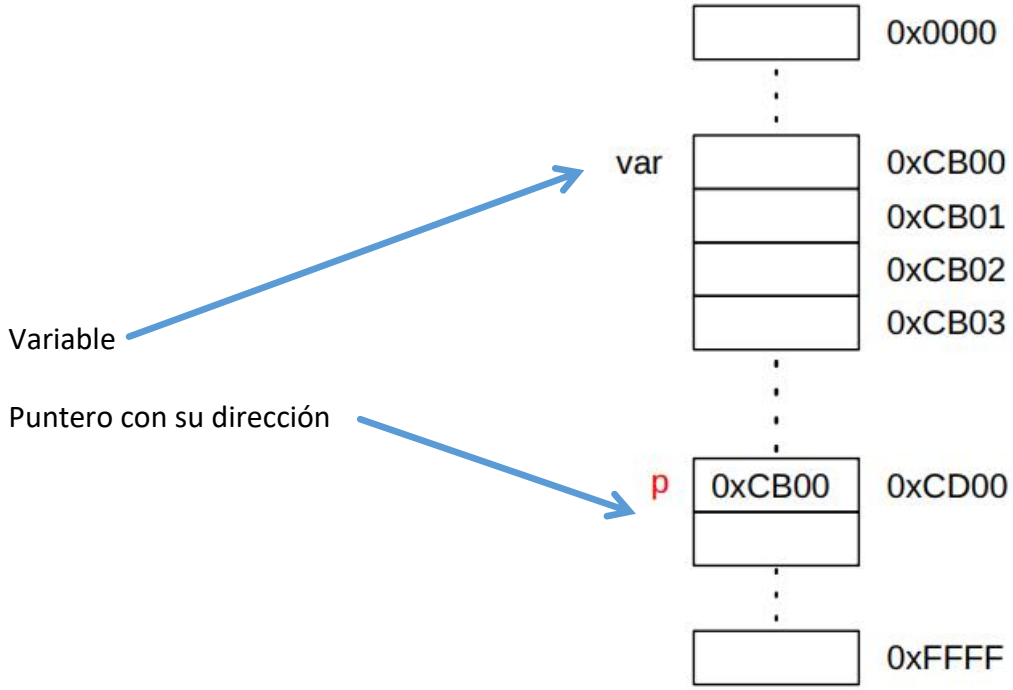
Búsqueda binaria:

Se selecciona un elemento central y se evalúa si es el elemento buscado, es mayor o menor. Se puede eliminar la mitad en la que no se encuentra, lo q lo hace muy veloz.

```c
int busqueda_binaria(int a[], int n, int key) {
    int middle;
    int low = 0;
    int high = n-1;
    int result = -1; // valor no encontrado

    while (low <= high && result == -1){
        middle = (low+high) / 2;

        if (key == a[middle])
           result = middle;

        if (key < a[middle])
           high = middle - 1;
        else
           low = middle + 1;
    }

    return result;
}
```

PUNTEROS

Los punteros son variables que almacenan la dirección de otra variable de su mismo tipo, solo pueden tener 3 tipos de valores: 0,NULL o una dirección. Se dice que “apuntan”a una variable cuando su valor es la posición de memoria de la misma.



Declaración:
Int var;
*Int *p;* (para declarar un puntero se usa (*) delante de su nombre, en este caso, el tipo "int" significa que apunta a una variable int). Los punteros pueden ser de cualquier tipo.

Para declarar varios en una misma sentencia de debe hacer:
char *p,*q; (cada uno debe tener *).

Operador (&):

p = &var;

El operador de dirección u operador de referencia, es un op. unario que devuelve la dirección de la variable que se coloque a continuación.
Independientemente de si *var* tenga o no un valor asignado, el puntero apunta a su dirección, y se puede acceder a la misma para leer o cargar valores.

Operador de desreferencia(*):

```
char var;
char *p;
p = &var;
*p = 21;
```

También llamado operador de indirección es un op. unario que se coloca delante de un puntero. Permite acceder indirectamente al contenido de la dirección de memoria apuntada. Puede ser tanto para cargar un valor o usarlo en alguna expresión:

```
printf("%d\n", *p);
```

Asignaciones válidas:

- Puede tener 3 tipos de valores: 0, NULL(cte. definida en stdio.h o en stddef.h).

- El 0 es el único entero que puede ser asignado a un puntero.
- Las direcciones solo pueden ser asignadas mediante “&” u otros punteros.
- NO se pueden hacer asignaciones de punteros de tipos incompatibles.
- Solo se puede asignar un puntero a otro si son del mismo tipo, excepto por los tipo *void,* el cual significa que no es de ningún tipo en particular.
- Se puede asignar cualquier puntero a un puntero “*void”.* Y se puede asignar un puntero *void* a cualquier tipo de puntero.

```
int var = 4;
char *p;
void *q;

q = &var;
p = q;
```

Se le pasa la dirección de un int a un tipo void, y luego este le pasa la dirección a un tipo char

- NO se puede desreferenciar un puntero void.

ARITMÉTICA DE PUNTEROS

Los punteros pueden ser operados por asignación,suma,restas, pre y post incremento/decremento y de incremento/decremento y asignación (+=, -=). No se pueden multiplicar o dividir.

Supongamos:

```
char vec[5];
char *p;
p = &vec[0];
p += 1;
```

(de esta manera se puede aprovechar el puntero para recorrer el arreglo)

Incrementa en 1 **byte** la dirección del puntero, por ej de 0xAB00 a 0xAB01, debido a que es un arreglo char, si fuera int, pasaría de 0xAB00 a 0xAB04

Relación entre punteros y arreglos:
En ocasiones los punteros y arreglos pueden ser intercambiados ya que el nombre del arreglo(sin los corchetes) es igual a la dirección del primer elemento.

Después de la línea p=&vec[0]; (cuando le pasamos la dirección del 1er elemento del arreglo), es posible acceder a sus elementos desreferenciando el puntero y haciendo un desplazamiento.
Ej:

```
*(p + 3) = 7;
```

(asigna un 7 al elemento vec[3])
Esto se conoce como notación puntero/desplazamiento.

También se puede usar el puntero con notación puntero/índice: `p[3] = 7;` (útil para las funciones)

Implementación de llamadas a función por referencia:
En C los llamados son siempre por valor, pero es posible hacerlo por referencia usando punteros, pasando la dirección de la variable a modificar en la función:

```c
void addone (int *p)
{
  *p += 1;
}

int main (void)
{
  int n = 3;

  addone(&n);
```

(incrementa la variable n a 4, ya que se le pasa la dirección de esta a la función y desreferenciando al puntero, accedemos al valor y lo incrementamos)

Calificador const:

Este calificador evita que la variable sea modificada, sin embargo, esto se podría hacer accediéndola directamente por su nombre, ya que la variable no fue definida con “const”.

```c
int main (void)
{
  int var;
  const int *p = &var;

  var = 42;
  printf("%d\n", *p);
```

Esta característica se usa en generalmente en funciones para asegurarse que los valores pasados no sean modificados.

```c
void copiar (const char *p, char *q)
{
  for (int i = 0; *(p+i) != 0; i++)
    *(q+i) = *(p+i);
}
```

En esa función “const char *p” asegura que la cadena apuntada por p no sea modificada por la función.

Formas de usar “const” en la lista de parámetros asociados a punteros:

- Puntero no constante a dato no constante:
  Se puede modificar el valor del puntero y de la variable a la que apunta.

```c
void carga_nota (int *pnotas, int n)
{
  int nota;
  for (int i=0; i < n; i++) {
    do {
      printf("Ingrese una calificación: ");
      scanf("%d", &nota);
    } while (nota<1||nota>10);
    *pnotas++ = nota;
  }
}
```

- Puntero no constante a dato constante:
  Se puede modificar el valor del puntero pero NO el de la variable a la que apunta.

```c
void imprimir (const int *pnotas, int n)
{
  for (int i=0; i < N; i++)
    printf("%d\n", *pnotas++);
}
```

  El puntero apunta a un tipo “const int” y solo se puede modificar el valor del puntero.

- Puntero constante a dato no constante:
  No se puede modificar el valor del puntero pero si el de la variable a la que apunta.

```c
void carga_nota (int * const pnotas, int n)
{
  int nota;
  for (int i=0; i < n; i++) {
    do {
      printf("Ingrese una calificación: ");
      scanf("%d", &nota);
    } while (nota<1||nota>10);
    *(pnotas+i) = nota; // *pnotas++ no está permitido
  }
}
```

- Puntero constante a dato constante:
  No se puede modificar ninguno de los valores (ni el puntero ni la variable).
  ```c
  const int * const pe;
  ```

Punteros a puntero

```c
int x, *p,**t;

x=5;
p=&x;//direccion de x
t=&p;//direccion de p
printf("%d\n",*t);//obtiene a p
printf("%d\n",**t);//obtiene a x
```

Si “p” apunta a “x” (int) y “t” apunta a “p”. Si desreferenciamos “t” una vez, obtendremos el valor de “p” (la dirección de memoria de x), si desreferenciamos “t” 2 veces, obtendremos el valor de x.
Entonces como t apunta a p y ambos son punteros del mismo tipo, entonces t es puntero a puntero.

Arreglos de caracteres:

Para almacenar cadena de caracteres se utilizan arrglos tipo “char”:

char saludo[ ] = “hola, mundo!”; (definido de esta manera, el arreglo tiene el tamaño justo para la cadena de texto + 1 caracter para el 0 al final de la línea)
saludo[6] = ‘M’; (se puede modificar la cadena siempre q no supere el tamaño original)

Punteros a cadenas constantes:

Otra manera de almacenar cadenas es con los punteros a cadenas de caracteres

*char *saludo = “hola,mundo!”;*

La diferencia es que no se pueden modificar.

Arreglos de cadenas:

```c
char numeros[5][10] = {"cero", "uno", "dos", "tres", "cuatro"};

for(int i = 0; i < 5; i++)
    printf("%s\n", numeros[i]);
```

Para guardar varias cadenas de caracteres se puede utilizar un arreglo bidimensional, donde 5(filas) es la cant. de cadenas que se puede almacenar, y 10(columnas), la cant. de caracteres que puede tener c/u como max.
La expresión “numeros[i];” devuelve la dirección de memoria de del primer elemento de la fila i.

c	e	r	o						
0xDA00	0xDA01	0xDA02	0xDA03	0xDA04	0xDA05	0xDA06	0xDA07	0xDA08	0xDA09
u	n	o							
0xDA0A	0xDA0B	0xDA0C	0xDA0D	0xDA0E	0xDA0F	0xDA10	0xDA11	0xDA12	0xDA13
d	o	s							
0xDA14	0xDA15	0xDA16	0xDA17	0xDA18	0xDA19	0xDA1A	0xDA1B	0xDA1C	0xDA1D
t	r	e	s						
0xDA1E	0xDA1F	0xDA20	0xDA21	0xDA22	0xDA23	0xDA24	0xDA25	0xDA26	0xDA27
c	u	a	t	r	o				
0xDA28	0xDA29	0xDA2A	0xDA2B	0xDA2C	0xDA2D	0xDA2E	0xDA2F	0xDA30	0xDA31

Arreglo de punteros:

Para almacenar varias cadenas también se podría usar un arreglo de punteros a caracteres.

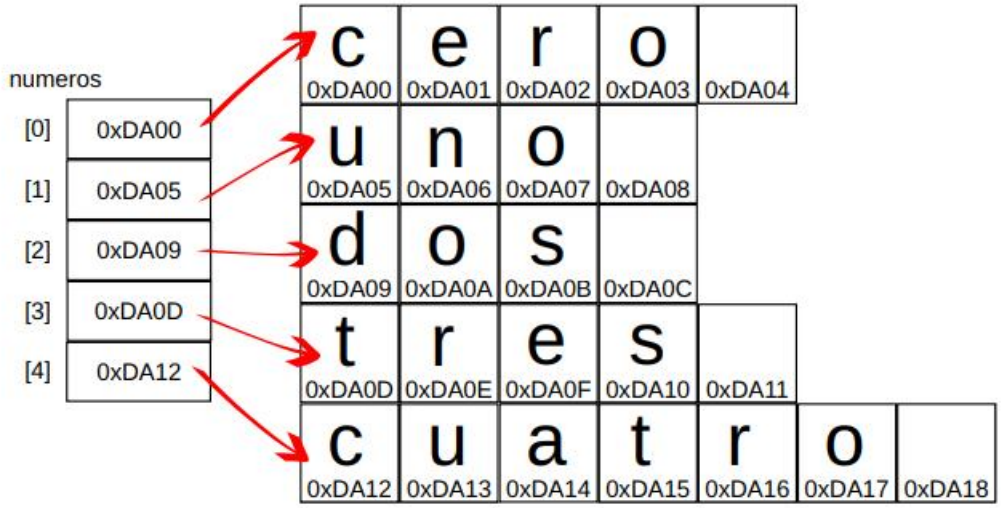
```
char *numeros[5] = {"cero", "uno", "dos", "tres", "cuatro"};

for(int i = 0; i < 5; i++)
    printf("%s\n", *(numero+i));
```

Esto es un arreglo de 5 elementos tipo puntero a char. Cada puntero apunta a una cadena de texto que no puede ser modificada..

Organización de datos con arreglos de punteros:

La organización es distinta cuando se usa este método:

c	e	r	o		u	n	o		d
0xDA00	0xDA01	0xDA02	0xDA03	0xDA04	0xDA05	0xDA06	0xDA07	0xDA08	0xDA09
o	s		t	r	e	s		c	u
0xDA0A	0xDA0B	0xDA0C	0xDA0D	0xDA0E	0xDA0F	0xDA10	0xDA11	0xDA12	0xDA13
a	t	r	o						
0xDA14	0xDA15	0xDA16	0xDA17	0xDA18	0xDA19	0xDA1A	0xDA1B	0xDA1C	0xDA1D
0xDA1E	0xDA1F	0xDA20	0xDA21	0xDA22	0xDA23	0xDA24	0xDA25	0xDA26	0xDA27
0xDA28	0xDA29	0xDA2A	0xDA2B	0xDA2C	0xDA2D	0xDA2E	0xDA2F	0xDA30	0xDA31



Punteros vs Arreglos multidimensionales:

La diferencia principal, es que en el caso de los arreglos, las expresiones:

```
numeros
numeros[0]
&numeros[0]
&numeros[0][0]
```

Hacen referencia a la **misma dirección**.

En caso de Arreglos de punteros:

```
char *numeros[5] = {"cero", "uno", "dos", "tres", "cuatro"};
```

el arreglo consta de 5 punteros, cada uno con un tamaño de 8 bytes.

Además, la dirección del primer elemento es igual a la dirección devuelta por la expresión que corresponde solo al nombre del arreglo, por lo tanto, *"numeros"* es igual a *"&numeros[0]"*.

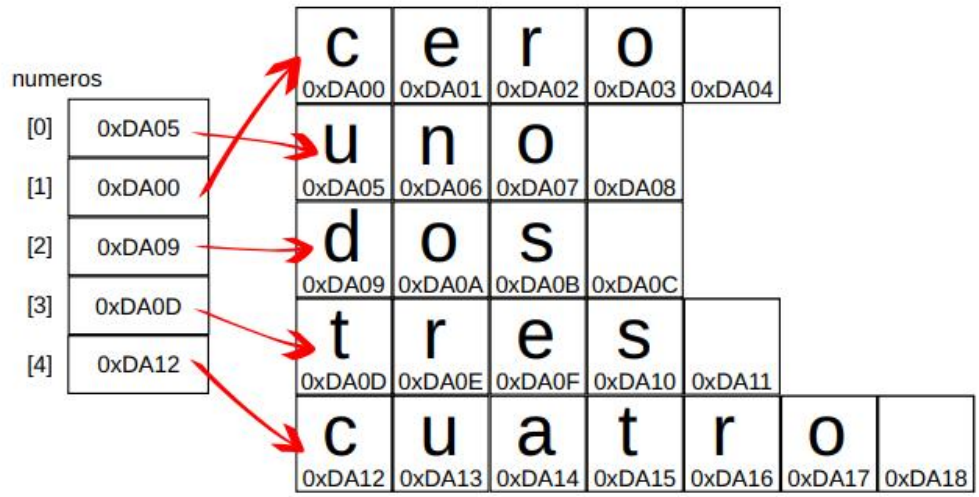
"*numeros[0]*" corresponde al primer elemento, pero en realidad contiene la dirección de memoria donde se encuentra la cadena. Entonces *"numeros[0]"* es igual a *"&numeros[0][0]"*.

Los 2 primeros son distintos de los segundos.

Ordenamiento de estructuras utilizando arreglo de punteros:

Se puede aprovechar que los punteros se pueden intercambiar para intercambiar filas de un arreglo en lugar de elemento por elemento en un arreglo bidimensional.

```
char *numeros[5] = {"cero", "uno", "dos", "tres", "cuatro"};
char *p = numeros[0];

numeros[0] = numeros[1];
numeros[1] = p;
```



Las cadenas no cambian su valor ni posición, cambia el orden de sus apuntadores.

Malloc, calloc, realloc y free

Así como la inicialización de variables reserva un lugar en la memoria (int = 4 bytes; char = 1 byte;etc) y asi como "*int *q = &var;*" define e inicializa un puntero apuntando a esa dirección. Es posible reservar memoria sin declarar una variable usando ***malloc*** declarada en **stdlib.h.**

```
int *p;
p = malloc(4);
```

Se llama con la cantidad de memoria que se necesita reservar como argumento, expresada en bytes. La función devuelve un puntero a la primera posición de la memoria reservada o un NULL si no pudo hacer la reserva.

El tamaño de un puntero(u otro tipo de dato) puede variar en base a la arquitectura (4 bytes en 32 bits, 8 bytes en 64 bits), para evitar errores en el tamaño pedido y poder usar el programa en distintas arquitecturas, se utiliza ***sizeof(int)*** o del tipo necesario. O también usando el puntero desreferenciado.

```
int *p;
p = malloc(sizeof (int));//tipo de dato
p = malloc(sizeof *p);//puntero desreferenciado
```

Cada vez que se reserve memoria con *malloc* debe liberarse una vez finalizado su uso. Para esto se utiliza **free**.

Se la debe llamara pasándole como argumento el puntero devuelto por malloc.

```
int *p;
p = malloc(sizeof (int));
free(p);//liberación de memoria
```

También se puede reservar memoria para un arreglo de tamaño n:

```
int n;
int *p;
printf("Ingrese cuantos elementos necesita: ");
scanf("%d", &n);
p = malloc(n * sizeof (int));
for (int i = 0; i < n; i++)
    *(p+i) = i*i;
for (int i = 0; i < n; i++)
    printf("%d ", *(p+i));
free(p);
```

Esto es más eficiente que declarar un arreglo sobredimensionado por las dudas, para luego terminar usando menos espacio. También es preferible antes que usar arreglos de dimensión variable.

Malloc no modifica el contenido de la memoria reservada. En cambio ***calloc,*** asigna memoria e inicializa en 0 todos los lugares, asegurándose de limpiar la memoria, también devuelve un puntero a la 1° posición de memoria reservada, y necesita 2 argumentos: la cant. de elementos y el tamaño de cada uno.

```
int n;
int *p;
printf("Ingrese cuantos elementos necesita: ");
scanf("%d", &n);
p = calloc(n, sizeof (int));
for (int i = 0; i < n; i++)
    *(p+i) = i*i;
for (int i = 0; i < n; i++)
    printf("%d ", *(p+i));
free(p);
```

En ocasiones es necesario cambiar el tamaño del arreglo. Para esto se usa ***realloc*** la cual puede redimensionar el arreglo, igual que el resto, devuelve un puntero a la posición de memoria reasignada. Pide 2 argumentos: el puntero de la memoria que se quiere modificar, y el nuevo tamaño.

TIPS:

- Siempre conviene chequear que el valor del puntero devuelto por alguna de estas funciones no sea NULL, y esto podría causar un error en tiempo de ejecución.
- Liberar por 2° vez un puntero da error en tiempo de ejecución.
- Desreferenciar un puntero que apunta a una memoria ya liberada es error en tiempo de ejecución.

PUNTEROS A FUNCIÓN

Los punteros también pueden apuntar a direcciones de memoria donde están implementadas las funciones. Estos pueden ser pasados a otras funciones o ser devueltos por las mismas. Pueden almacenarse en arreglos y asignarse a otros punteros a funciones.

Deben ser definidos antes de su uso. En las declaraciones debe aclararse que tipos de valores recibirá la función apuntada y qué tipo devolverá,

`int (*p)(int, int);` esta es la definición de un puntero a una función que recibe 2 enteros y devuelve un entero.

Prototipo de función común que devuelve un puntero a entero	**Puntero a función que devuelve un entero**
int *funcion(int,int);	int (*pfuncion)(int,int);

Como los arreglos, el nombre de la función indica la posición de memoria donde comienza su implementación. Entonces si se tiene una definición:

```
int add (int a, int b) {
    return a+b;
}
```

Se puede hacer una asignación como:

```
int (*p)(int, int);
p = add;
```

Llamado de funciones apuntadas por medio de la desreferencia:

```c
int (*p)(int, int);
p = add;
printf("%d\n", (*p)(5,3));
```

En lugar de usar el nombre de la función, se desreferencia el puntero y se pasa los argumentos encerrados entre paréntesis

```c
#include <stdio.h>


int suma (int a, int b) {
    return a+b;
}
int resta (int a, int b) {
    return a-b;
}
int main(void) {

    int (*p)(int,int);
    int r;

    p = suma;
    r = (*p)(5,4);
    printf("%d\n", r);

    p = resta;
    r = (*p)(5,4);
    printf("%d\n", r);

    return 0;
}
```

r = (*p)(5,4); podría escribirse:

r = p(5,4);

Pero no es aconsejable porque se podría confundir con una función no definida. Se aconseja usar la versión con el operador de desreferencia.

Ejemplo de ordenamiento:

Se quiere ordenar un arreglo de enteros, de manera ascendente o descendente según se pida. Primero se define la función donde se van a intercambiar los valores:

```c
void swap(int *p, int *q) {
    int temp = *p;
    *p = *q;
    *q = temp;
}


void ascendente (int *p, int *q) {
    if (*p > *q)
        swap(p, q);
}
void descendente (int *p, int *q) {
    if (*p < *q)
        swap(p, q);
}
```

Declaramos la función que va a intercambiar los valores

Declaramos las funciones para cada caso,donde solo cambia el condicional, en p y q estarán los valores de los elementos consecutivos a analizar

```c
void burbuja(int arr[], int n, void (*compara)(int*,int*)) {
    for (int j = 0; j < n-1; j++)
        for (int i = 0; i < n-1-j; i++)
            (*compara)(&arr[i], &arr[i+1]);

}
```

La función clave es donde se realiza el método burbuja, recibe como argumentos: el arreglo, su tamaño, y la función a ejecutar (ascendente o descendente)

```c
int main(int argc, char *argv[]) {

    int vec[N] = {10,0,4,8,68,123,3,5,7,2};

    burbuja(vec, N, descendente);
    imprimir(vec, N);

    return 0;
}
```

En el main simplemente se llama a la función burbuja, se declara el arreglo y se agrega la función para imprimir los elementos.

Arreglos de punteros a función

Como los demás tipos de variables, se puede hacer un arreglo de punteros a función.

```c
int (*p[4])(int,int);
```

Si tenemos 4 funciones definidas como:

```c
int suma (int a, int b) {
    return a+b;
}
int resta (int a, int b) {
    return a-b;
}
int mult (int a, int b) {
    return a*b;
}
int divi (int a, int b) {
    return b!=0?a/b:0;
```

Entonces:

```c
int (*p[4])(int,int);
int r;
p[0] = suma;
p[1] = resta;
p[2] = mult;
p[3] = divi;
for (int i = 0; i < 4; i++) {
    r = (*p[i])(6,2);
    printf("%d\n", r);
}
```

Se trata de un arreglo de 4 punteros que apuntan a su vez a 4 funciones que reciben 2 enteros y devuelven 1 entero. En el *for* se llama a cada una de estas funciones pasándoles como argumento los n° 6 y 2.

También se puede inicializar en la definición del arreglo:

```c
int (*p[])(int,int) = {suma, resta, mult, divi};
```

Argumentos por línea de comandos

Ver: https://www.youtube.com/watch?v=LGuERz2KhZE

Estructuras y uniones en C + Campos de bits

Conceptos a considerar:

Int,char,float,void y los punteros son conocidos como tipos *escalares.* Los arreglos son de tipos *agregados*, sirven para guardar datos relacionados del mismo tipo bajo un mismo nombre.

Estructuras:

Son de tipos de datos *derivados* que guardan datos relacionados de distinto tipo.

Definición:

```
struct dato {
    int a;
    char b;
};
```

- *struct* es la palabra reservada para indicar que es una estructura.
- *dato* es la etiqueta de la misma.
- Entre llaves se definen los *miembros* de la estructura (cualquier cantidad), pueden ser de cualquier tipo (int,char,etc), además pueden ser arreglos, punteros u otras estructuras.
- La definición de la misma termina en (;).

La definición de una estructura no asigna memoria, sino que crea un **nuevo tipo de dato** que se puede usar para definir variables:

```
struct dato d;
```

Acá se define la variable "d" de tipo "*struct dato".* Las variables de tipo "struct dato" solo deberán hacerse luego de definir la estructura.

Se pueden definir variables en la misma definición de la estructura:

```
struct dato {
    int a;
    char b;
} d ;
```

En este caso, "d" es una variable global.

Inicialización:

Al igual que los arreglos, se inicializan entre llaves, donde los elementos se asignan en el orden en que están definidos en la estructura.

```
struct dato d ={1,'a'};
```

Si se desea inicializar todos lo elementos en 0, simplemente se coloca el mismo entre las llaves: { 0 }. Si hay menos inicializadores que miembros, los que faltan son puestos en 0. Si hay más inicializadores que miembros, genera un error de compilación.

También se puede inicializar un elemento usando el operador punto y el miembro que corresponde. Se conocen como *inicializadores designados*:

```
struct dato d = {.b='a'};
```

Operador punto:

Así como en los arreglos se usa [ ] para identificar cada elemento del mismo, en el caso de la estructura se usa el op. punto y el nombre del miembro al que se quiere acceder.

```
struct dato d ={1,'a'};

printf("miembro a: %d\n", d.a);
printf("miembro b: %c\n", d.b);
```

"d" al ser del tipo struct dato, tiene miembros a y b a los que se accede con el operador punto. (d.a accede al miembro "a" y así con el resto de miembros que pueda tener una estructura).

Acceso a los miembros:

```c
struct persona {
    int dni;
    char nombre[80];
} ;

int main(int argc, char *argv[]) {

    struct persona gente ={2548756,"nombre genérico"};

    gente.nombre[0]='N';
    printf("Nombre: %s", gente.nombre);
```

Operaciones permitidas:

- Asignación

```c
struct punto_2d {
    float x;
    float y;
};
int main (void) {
    struct punto_2d p1, p2 = {3,2};

    p1 = p2;

    printf("(%.2f, %.2f)\n", p1.x, p1.y);

    return 0;
}
```

Se puede hacer asignación solo si se trata del mismo tipo de estructuras, sino hay error de compilación.

- Tomar dirección de memoria (&)
- Desreferenciar (*)
- Acceder a miembros con . o ->
- Operador *sizeof*

Operaciones no permitidas:

Operadores de relación (==,!=,>,<,etc.).

Punteros a estructuras

A diferencia de con variables hay que recordar usar el tipo de dato “struct”:

```c
struct punto_2d p1, p2 = {3,2};
struct punto_2d *pp;

pp = &p1;

printf(".2f",(*pp).x);
```

En la expresión (*pp).x deben usarse los paréntesis debido al orden de precedencia, para que la desreferencia se realice primero. De lo contrario da error porque el op. punto espera una estructura y un miembro al que acceder, no un puntero.

Operador flecha (->):

Para simplificar la notación “(*pp).x” se usa este operador de forma que la expresión quede: `printf(".2f",pp->x);`

El operador flecha espera un puntero a una estructura a la izquierda y el miembro de la misma a la derecha.

Arreglos de estructuras

Para definir un arreglo de estructuras simplemente se usa el corchete en el nombre de la variable:
`struct punto_2d puntos[5];` este es un arreglo de 5 elementos, c/u de tipo “struct punto_2d”.

Para acceder a los miembros de cada estructura:

```c
#include <stdio.h>
struct punto_2d {
    float x;
    float y;
};
int main (void) {

    struct punto_2d puntos[5] = { {0,0},{6,7} };

    puntos[0].x = 3; puntos[0].y = 3;
    puntos[3].x = 4; puntos[3].y = 5;

    for(int i=0; i<5 ; i++){
        printf("(%.2f, %.2f)\n", puntos[i].x, puntos[i].y);
    }
    return 0;
}
```

Se opera primero el corchete por estar más a la izquierda. Como es un arreglo, para accesar a un elemento se usa los corchetes con el índice correspondiente. Como el elemento es de tipo “struct punto_2d” este posee miembros, los cuales son accesados luego con el op. punto.

Arreglos asignados dinámicamente:

```c
struct personal {
    int dni;
    char nombre[80];
    int legajo;
};
int main (void)
{
    struct personal *p;
    int n=2;
    p = malloc (n*sizeof (struct personal));

    for(int i = 0; i < n; i++) {
        printf("Ingrese Nombre: ");
        scanf(" %80[^\n]s", (p+i)->nombre);
        printf("Ingrese DNI: ");
        scanf("%d", &(p+i)->dni);
        printf("Ingrese Legajo: ");
        scanf("%d", &(p+i)->legajo);
    }

    for(int i = 0; i < n; i++) {
        printf("Nombre: %s\n", (p+i)->nombre);
        printf("DNI: %d\n", (p+i)->dni);
        printf("Legajo: %d\n", (p+i)->legajo);
    }
    free(p);

    return 0;
}
```

Funciones y estructuras:

A las funciones se pueden pasar: miembros de la estructura, la estructura completa, un puntero a una estructura o un arreglo de estructuras.

En el primer y segundo caso, el pasaje es por valor, o sea que la estructura original no se modifica.

PARA PASAR MIEMBROS:

```
int suma (int a, int b) {
    return a+b;
}
 printf("%d",suma(p.x,p.y));
```

PASAR ESTRUCTURAS COMPLETAS:

Funciona igual que con cualquier variable

```
struct punto_2d{
    int x;
    int y;
};
int suma (struct punto_2d a, struct punto_2d  b) {
    return (a.x+b.x)-(a.y-b.y);
}
int main(int argc, char *argv[]) {
    struct punto_2d p = {9,5}, p2 = {23,43};

    printf("%d",suma(p,p2));
    return 0;
}
```

PASAR ESTRUCTURAS COMO LITERALES:

Pasar estructuras como literales es como pasarle a una función un número en vez de una variable. En este caso en los argumentos de la función se coloca entre paréntesis el tipo de dato y entre llaves los valores de cada miembro. Se conocen como *literales compuestos*. Tambien se pueden usar los inicializadores designados.

```
struct punto_2d{
    int x;
    int y;
};
int suma (struct punto_2d a, struct punto_2d  b) {
    return (a.x+b.x)-(a.y-b.y);
}
int main(int argc, char *argv[]) {
    struct punto_2d p = {9,5};

    printf("%d",suma(p,(struct punto_2d) {8,5}));
    return 0;
}
```

FUNCIONES QUE DEVUELVEN ESTRUCTURAS:

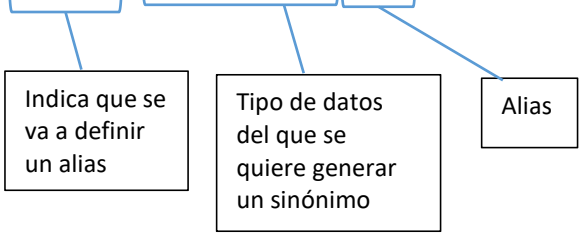
Cuando se pasan arreglos de estructuras a funciones, son automáticamente por referencia como todos los arreglos, pero si la estructura tiene arreglos, estos se pasan por copia. Si bien las funciones pueden devolver un solo valor, también pueden devolver **una** estructura, de esta manera se pueden devolver varios valores.

```c
#include <stdio.h>

struct persona {
    int dni;
    char nombre[80];
};

struct persona carga (void) {

    struct persona r;

    printf("Ingrese su DNI: ");
    scanf("%d", &r.dni);
    printf("Ingrese su nombre: ");
    scanf("%s", r.nombre);
    return r;
}
int main(void){

    struct persona per;

    per = carga();

    printf("%d\n", per.dni);
    printf("%s\n", per.nombre);

    return 0;
}
```

TYPEDEF

La palabra clave typedef provee un mecanismo para generar sinónimos o alias.

```c
typedef unsigned int uint;
```



A partir de este punto se puede usar indistintamente el alias o el tipo completo:

```c
unsigned int valor1;
uint valor2;
```

Se usa mucho en estructuras para simplificar notación, supongamos la estructura:

```c
struct punto_2d{
    float x;
    float y;
};
```

El prototipo de funciones que reciben estrucutras con este nombre podrían ser extensas:

```
struct punto2D suma (struct punto2D p, struct punto2D q);
```

Se puede usar typedef de varias formas:

```
struct punto_2d{
    float x;
    float y;
};
typedef struct punto_2d p2D;
```

O

```
typedef struct punto_2d{
    float x;
    float y;
} p2D;
```

Con esto el prototipo de función pasa a ser mucho más corto:

```
p2D suma (p2D p, p2D q);
```

Las estructuras pueden ser definidas sin etiquetas si se definen usando typedef:

```
typedef struct {
    float x;
    float y;
} punto2d;
```

O se usan para definir una variable global(si se utiliza este método no se van a poder crear más variables de este tipo)

```
struct {
    float x;
    float y;
} p2d;
```

Estructuras autoreferenciadas (no entra en parcial práctico):

Para estructuras de datos complejos en C, se llaman así porque 1 o más de sus miembros es un puntero apuntando a una estructura del mismo tipo que se está definiendo:

```
struct nodo {
    int dato;
    struct nodo *p;
};
```

Si se tiene definida así una estructura se puede asignar memoria suficiente para contenerla:

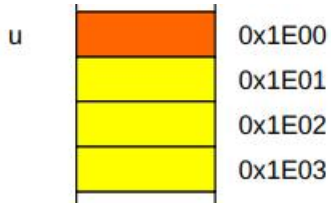
```
struct nodo *p_lista;
p_lista = malloc(sizeof(struct nodo));
```

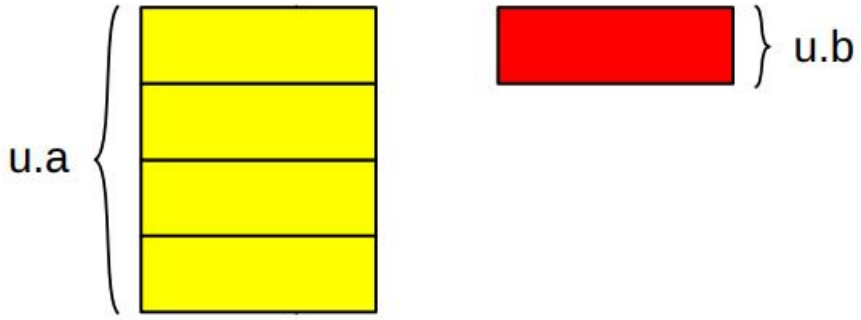
Uniones

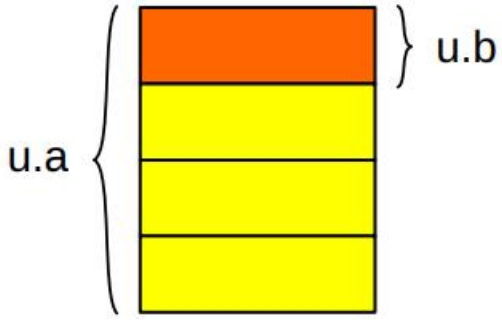
De tipos derivados y de similar definición que las estructuras. Tiene miembros como las estructuras y se acceden con (.) o (->), la palabra clave acá es *union.*

```
union dato {
    int a;
    char b;
};

union dato u;
```

La diferencia es que cada miembro de la unión comparte la posición de memoria con los otros:

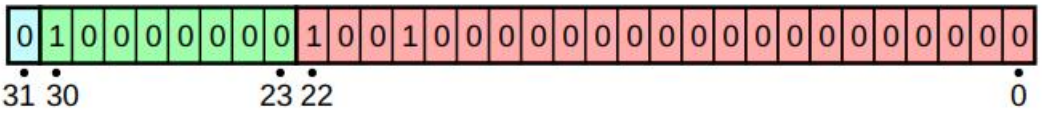




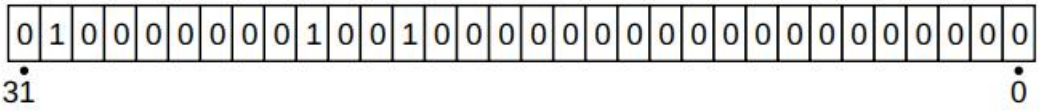


El tamaño en memoria de esta es suficiente para almacenar el miembro que ocupe mayor espacio. Compartir memoria puede ser un problema si no se accesa correctamente a los miembros.

Si se tiene un número real, por ej. `3.125` almacenado en formato IEEE754 de simple precisión, en bits sería



pero si se interpretan esos mismos bits



como un entero... se lee `1078460416` ya que

$2^{30}+2^{22}+2^{19} = 1078460416$

Entonces, si se asigna un valor a un miembro de la unión, este se guarda siguiendo las reglas de ese tipo de dato. Pero como se guardan en bits, si se lee desde un miembro de tipo diferente se interpretan los bits con las normas de ese tipo de datos.

Inicialización:

Solo se puede inicializar un valor, ya que es compartido por todos los miembros. El formato de carga depende de cual sea el valor que esté primero en la definición de la unión.

```
union int_float u = {45};//para el primer miembro
union int_float u = {.real=45.76};//para el segundo miembro
```

Operaciones permitidas:

- Asignar uniones a otras uniones del mismo tipo.
- Se puede usar (&) para devolver la dirección de memoria de la unión (igual a cualquiera de sus miembros).
- El op. punto y el op. flecha se usan como en las estructuras.

Operaciones NO permitidas:

- Comparaciones (==,!=,etc).
- Operaciones aritméticas (+,-,*,/)

Posibles usos:

Pueden tener arreglos como miembros (pasados por valor a las funciones), puede ser de cualquier tamaño ya que el tamaño de la unión será igual a la de su miembro mayor.

Se pueden hacer arreglos de uniones:

```
union int_float u[10];
```

Pueden tener estructuras como miembros y viceversa:

```
union numero {
    int entero;
    float real;
    struct {
        short int r;
        short int i;
    } complejo;
};

struct int_float {
    int es_entero;
    union {
        int entero;
        float real;
    } num;
};
```

Ahorro de memoria:

```c
#include <stdio.h>
#include <string.h>
union mensajes {
    char info[80];
    char aviso[80];
    char error[80];
};
int main (void)
{
    union mensajes msg;
    strcpy(msg.info, "Uso para ahorro de memoria");
    printf("%s\n" , msg.info);
    printf("Uso de memoria: %lu bytes\n", sizeof msg);
    return 0;
}
```

Comunicación en serie (ver ejemplo diapositiva 83, unidad 9)

Campos de bit

Las variables en general, y las estrucutras en particular, pueden ser capaces de contener valores más grandes de lo que necesitan.

```c
struct time {
    unsigned int h; // 0-23
    unsigned int m; // 0-59
    unsigned int s; // 0-59
};
```

El rango de unsigned int es 0 a 4294967295.

```
Tamaño de la estructura: 12 bytes
```

Definición:

```c
struct time {
    unsigned int h:5;
    unsigned int m:6;
    unsigned int s:6;
};
```

- Se usan (:) para definir cuantos bits se deben usar.
- El tamaño de bits no puede exceder el tamaño del tipo usado.
- Solo se pueden declarar miembros *int* o *unsigned int.*
- El tamaño de bits puede ser 0.
- No se puede usar (&) para tomar la dirección de los bits.
- La organización de un campo de bits en la memoria depende del compilador.

También existen los campos de bits sin nombre que se utilizan para agregar *padding* (completar lugares para alinear datos en la memoria).

```c
struct time {
    unsigned int h:5;
    unsigned int :27;
    unsigned int m:6;
    unsigned int s:6;
};
```

También se puede poner ancho 0 a un campo de bit sin nombre, esto genera el padding necesario para alinear el siguiente campo de bit en la siguiente unidad de almacenamiento.

Inicialización y acceso a un campo de bits:

```
struct time t = {23, 20, 59};
printf("%d:%d:%d\n", t.h, t.m, t.s);
```

Se inicializan entre llaves como las estructuras, incluso con inicializadores designados.
El acceso se realiza mediante el operador punto/flecha(si es un puntero).

Constantes de enumeración

Son un conjunto de constantes enteras representadas por identificadores, su definición es semejante a las estructuras.

```
enum dias_laborales {
    LUNES,
    MARTES,
    MIERCOLES,
    JUEVES,
    VIERNES
};
```

Ha no ser que se explicite lo contrario, se enumeran las ctes desde el 0.

- La palabra clave es *enum*.
- Sigue la etiqueta(opcional).
- Las ctes van separadas por comas.
- Se recomienda el uso de mayúsculas.

En la definición se puede asignar cualquier valor entero usando el “=”, a partir de ahí, los siguientes se incrementan de a 1.

```
enum colores {
    ROJO, // 0
    VERDE, // 1
    AZUL = 10, // 10
    BLANCO = 20, // 20
    AMARILLO, // 21
    NARANJA = 5, // 5
    NEGRO // 6
};
```

Usos:

```
#include <stdio.h>

enum dias {LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO};

char *nombre_dia[] = {
    "lunes", "martes",
        "miércoles", "jueves",
        "viernes", "sábado",
        "domingo",
};
int main (void)
{
    enum dias hoy;
    for (hoy = LUNES; hoy <= DOMINGO; hoy++ )
        printf("Hoy es %s\n", nombre_dia[hoy]);
    return 0;
}
```

Operadores a nivel de bit

Las computadoras representan toda la información interna usando bits.

Los bits solo pueden tener valores 0 y 1. Los operadores a nivel de bit realizan las operaciones bit a bit entre sus operandos.

```
char a = 77; // 0 1 0 0 1 1 0 1
char b = 42; // 0 0 1 0 1 0 1 0
```

Operadores lógicos:

Tablas de verdad:

AND:

a_i	b_i	&
0	0	0
0	1	0
1	0	0
1	1	1

```
a = 77;          // 0 1 0 0 1 1 0 1
b = 42;          // 0 0 1 0 1 0 1 0
c = a & b;       // 0 0 0 0 1 0 0 0
```

OR:

a_i	b_i	|
0	0	0
0	1	1
1	0	1
1	1	1

```
a = 77;          // 0 1 0 0 1 1 0 1
b = 42;          // 0 0 1 0 1 0 1 0
c = a | b;       // 0 1 1 0 1 1 1 1
```

OR exclusiva (XOR):

a_i	b_i	$\wedge$
0	0	0
0	1	1
1	0	1
1	1	0

```
a = 77;          // 0 1 0 0 1 1 0 1
b = 42;          // 0 0 1 0 1 0 1 0
c = a ^ b;       // 0 1 1 0 0 1 1 1
```

COMPLEMENTO:

a_i	$\sim$
0	1
1	0

```
a = 77;          // 0 1 0 0 1 1 0 1
c = ~a;          // 1 0 1 1 0 0 1 0
```

Operadores de desplazamiento:

Los operadores << y >> son operadores que generan un desplazamiento sobre los bits de un int o un char. El operando de la izq. es el que se ve afectado por el corrimiento. El de la derecha indica cuantos bits de corrimiento.

```
a = 5;           // 0 0 0 0 0 1 0 1
c = a<<1;        // 0 0 0 0 1 0 1 0
```

Cuando se usa el operador << los bits se corren a la izquierda, agregando 0 (ceros) a la derecha.

Cuando se usa el operador >> los bits se corren a la derecha, agregando 0 (ceros) a la izquierda.

En caso de ser un número negativo pueden pasar dos cosas...Se agregan unos (el número sigue siendo negativo) o se agregan ceros (cambia el signo). Depende de la arquitectura.

Operadores y asignación:

`&=`	AND y asignación
`|=`	OR y asignación
`^=`	XOR y asignación
`<<=`	Desp. izq. y asignación
`>>=`	Desp. der. y asignación

TABLA DE PRECEDENCIA FINAL

Operador	Asociatividad
() [] . ->	Izq. a Der.
+ - (tipo) ++ -- ! ~ & *	Der. a Izq.
* / %	Izq. a Der.
+ -	Izq. a Der.
<< >>	Izq. a Der.
< <= > >=	Izq. a Der.
== !=	Izq. a Der.
&	Izq. a Der.
^	Izq. a Der.
|	Izq. a Der.
&&	Izq. a Der.
||	Izq. a Der.
? :	Der. a Izq.
= += -= /= *= &= |= ^= <<= >>= %=	Der. a Izq.
,	Izq. a Der.