❑ CONCEPTO DE **VECTOR** O **ARRAY**

✓ Concepto

Vector, **arreglo** o **array**: Objeto que permite definir un almacenamiento (con un nombre común) en memoria principal de una colección de datos:

- del mismo tipo
- ordenados de manera secuencial (en posiciones contigüas de memoria), de manera que cada dato ocupa una posición determinada (0, 1, 2 …) denominada ***índice***. La forma de acceder a un dato determinado es indicar su número de posición.

Ejemplo

- Notas de los alumnos de una clase ordenadas por número de orden del alumno

Índice	0	1	2	3	4	5	6
NOTAS	7.5	2	6	9	2	4	6

NOTAS [3]=9

- Número de que se obtiene en cada una de las tiradas de un dado

Índice	0	1	2	3	4	5	6
TIRADAS	5	6	2	6	5	1	2

TIRADAS [**6**]=2

✓ Declaración

tipo_dato identificadorV[dimensión]

El ***tipo*** indica al compilador cuantos bytes se necesitan para cada elemento y la ***dimensión*** cuantos elementos. Es típico definir la dimensión con ***#define dimensión N***

- Se puede inicializar un vector al declarlo
- Solo se deberían declarar sin dimensión los vectores de caracteres inicializados:

  char vector[]="hola"

✓ Acceso

identificadorV[indice]

Donde indice es una variable int o char (numérica)

- Un array indexado se utiliza exactamente igual a como se utilizaría una variable del tipo base del vector (asignarle un valor,usarla en expresiones)
- No se comprueba que el índice está dentro de los límites ('0' y 'dimensión-1'): es responsabilidad del programador evitar escribir o leer fuera de los limites.

- Escribir fuera de los límites de un array es especialmente pernicioso: se altera el contenido de posiciones de memoria que pueden contener cualquier cosa (otras variables, código máquina, etc).

```
int main(int argc, char * argsv[])
{
 unsigned char
 nDiasxMes[12]={31,28,31,30,31,30,31,31,30,31,30,31};
 int mes;

 /*¿Cuántos días tiene Enero?*/

printf("El número de dias de Enero es%i",
nDiasxMes[0];


      /*¿Cuantos días tiene un mes cualquiera?*/

 printf("Indica un número de mes entre 1 y 12");
 do{
   scanf("%i",&mes);
 }while((mes<1) || (mes>12));

 printf("El número de dias del més número %i es %i",
                                          mes,
nDiasxMes[mes-1];

 /*¿Cuantos días tiene Diciembre?*/
```

```
 printf("El número de dias de Diciembre es %i",

nDiasxMes[11];


}
```

❑ ARRAYS Y FUNCIONES

- Un array declarado como variable local de una función se usa igual que cualquier otra variable

- Un array que aparezca como parámetro de una función presenta algunas particularidades:

  1-Llamada a una función que tenga como parámetro un array: se escribe el nombre del array sin corchetes

  2-Declaración como parámetro formal en la cabecera de una función: se escribe **tipo nombreVector []** (sin escribir un tamaño, aunque no sería incorrecto).

  3- Los vectores no se pueden devolver en una función (return), por lo que si se desea que una función modifique un array en principio tendríamos que pasarlos como parámetro por referencia. Esto no es necesario ya que los vectores se pasan siempre por referencia (son en realidad punteros).

Por tanto cualquier cambio que haga una función en un array pasado como parámetro será un cambio permanente. Un error típico del programador en C novato es confiar que un array pasado como parámetro actual a una función no se modificará.

Tampoco es necesario utilizar el operador & delante del nombre del array . El nombre de un array equivale sintácticamente a la dirección del elemento cero (es decir array == &array[0]).

```
#include <stdio.h>

/*Declaración alternativa: 'void visualizar(int *)
*/

/*prototipo: Declaración sin poner nombre del
array*/
void visualizar(int []);

int main() /* rellenamos y visualizamos */
{
 int array[25],i;
 for (i=0;i<25;i++)
 {
       printf("Elemento nº %d",i+1);
       scanf("%d",&array[i]);
 }
 visualizar(array);  //o también:
visualizar(&array[0])
```

```
}

/*Declaración alternativa 'void visualizar(int * array)*/

/*Implementación: obligatorio poner nombre al array */
void visualizar(int array[])

{
  int i;
  for (i=0;i<25;i++) printf("%d",array[i]);
}
```

❑ MANIPULACIONES BÁSICAS DE UN VECTOR O ARRAY

➘ Inicialización

Si el tamaño del array no es muy grande, y sabemos de antemano los valores que debe tener

```
int vector[3]={1,2,3};
char vector[5]={'h','o','l','a','\0'};
char vector[5]="hola";
```

➘ Recorrido

Recorrido secuencial del array: Siempre se debe utilizar un bucle para desde ***0*** hasta el ***tamaño - 1***. Por ejemplo, recorrer el array para inicializarlo con 0 (tarea que, en principio, no tiene mucho sentido)

```
void inicializa (int vector[])
{
  int i;
 for (i=0; i<DIM; i=i+1)
      vector[i]=0;
}
```

➘ Imprimir el contenido de un vector o array

```
void imprime (int vector[])
{
 int i;
 for (i=0; i<DIM; i=i+1)
       printf("%d", vector[i]);
}
```

➘ Lectura de los datos en un vector o array

```
void lee (int vector[])
{
 int i;
 for (i=0; i<DIM; i=i+1)
 {
   printf("dato: ");
  scanf("%d", &vector[i]);
 }
}
```

➘ Ejemplo de trabajo con vectores o arrays

```
    #define DIM 10 //Tamaño del vector

    void inicializa(int vector[]);
    void imprime (int vector[]);
    void lee (int vector[]);

    int main()
    {
     int vector[DIM];

     inicializa(vector);     //No es necesario
     imprime (vector);
     lee(vector);
     imprime (vector);
    }
```

❑ MATRICES:
ARRAYS MULTIDIMENSIONALES

✓ Concepto

Los arrays pueden tener 1 dimensión (los estudiados antes) o 2 (arrays bidimensionales o matrices), 3 o mas.

En una matriz:

- Se distribuye la información en forma de tabla por filas y columnas.
- Las filas y columnas se indexan desde 0 en adelante.
- En la intersección de una fila y una columna hay un solo dato (identificado por num_fila y num_ columna)

Las matrices suelen utilizarse para relacionar dos magnitudes, como por ejemplo, la nota de un determinado alumno en una determinada asignatura.

Boletin[Nasig][Nalum]

	0	1	2	3
0	5	6	9	2
1	10	6	5	5
2	7	4	6	4
3	8	5	5	1

El alumno 1 en la asignatura 2 tiene un 4
Boletin[2][1]=4

✓ Declaración

tipo_dato identificadorM[dimensiónF][dimensiónC]

Por ejemplo:

```
#define ALUMNOS 30
#define ASIGNATURAS 4

float notas[ALUMNOS][ASIGNATURAS];
```

✓ Acceso

identificador [fila] [columna]

Para acceder a una posición de una matriz deberemos usar dos variables enteras 'fila', 'columna' que deberían tomar valores desde 0 hasta dimensiónF-1 y 0 hasta dimensionC-1 respectivamente.

Por ejemplo:

```
int fila=1, columna=2;
notas[fila][columna]=7.5;
```

❑ MATRICES Y FUNCIONES

- Una matriz declarada como variable local de una función se usa igual que cualquier otra variable

- Una matriz que aparezca como parámetro de una función presenta algunas particularidades:

  1-Llamada a una función que tenga como parámetro una matriz: escribir nombre de matriz sin corchetes ni &

  2-Declaración como parámetro formal en la cabecera de una función se escribe sin tamaño para las filas y a continuación [dimensión] para las columnas:

tipo nombreMatriz [][Columnas]

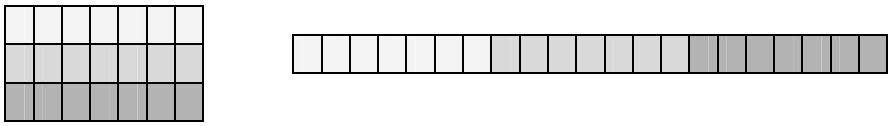
En general, si el argumento es una matriz n-dimensional es obligatorio especificar todas las dimensiones menos la primera

Tipo nombreMatriz [][DIM2][DIM3]

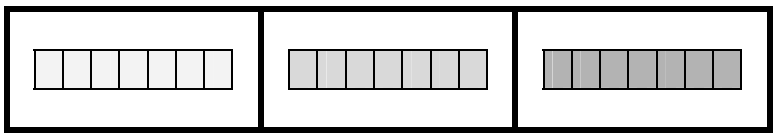
La razón es que una matriz se almacena realmente en memoria "por filas".

Por ejemplo, en el caso bidimensional:

int matriz[3][7] En memoria:



Puede verse así: El vector multidimensional es un vector de dimensión DIM1 en el que en cada posición hay a su vez un vector de dimensión DIM2, y asi sucesivamente:



Para que el compilador pueda indexar correctamente la matriz y, por ejemplo acceder a matriz[3][4], necesita conocer DIM2:

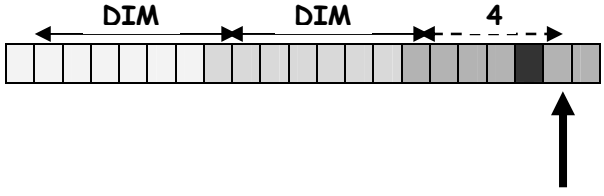
matriz[3][*4*]

==

posición de la Fila **3**, Columna *4*

==

posición Nº: **3***DIM2+*4*



❑ MANIPULACIONES BÁSICAS DE UNA MATRIZ

➘ Inicialización

Si el tamaño del array no es muy grande, y sabemos de antemano los valores que queremos almacenar en él, podemos inicializarlo en el momento de su declaración, igualándolo a la lista de valores separados por comas y encerrados entre llaves { }.

```
int m[3][3]= { {1,2,3},   /*fila 0*/
               {4,5,6},  /*fila 1*/
               {7,8,9},   /*fila 2*/
             };
```

➘ Recorrido de una matriz

Para recorrer una matriz secuencialmente: dos bucles para anidados desde 0 hasta la dimensión menos 1. Por ejemplo, si queremos inicializar todas las casillas a por ejemplo, a cero (tarea en principio absurda);

```
void inicializa (int m[][COLUMNAS])
{
      int i,j;
      for (i=0;i<FILAS;i=i+1)
            for (j=0;j<COLUMNAS;j=j+1)
                  m[i][j]=0;
}
```

➘ Imprimir el contenido de una matriz

```
void imprimir (int m[][COLUMNAS])
{
      int i,j;
      for (i=0;i<FILAS;i=i+1)
            for (j=0;j< COLUMNAS;j=j+1)

printf("m[%d,%d]=%d",i,j,m[i][j]);
}
```

➘ Lectura de los datos de una matriz

```
void lee (int m[][COLUMNAS])
{
      int i,j;
      for (i=0;i<FILAS;i=i+1)
            for (j=0;j< COLUMNAS;j=j+1)
            {
                  printf("dato[%d,%d]:
",i,j);
                  scanf("%d",&m[i][j]);
            }
}
```

➘ Ejemplo de trabajo con matrices

```
#define FILAS 3
#define COLUMNAS 4

void inicializa(int m[][COLUMNAS]);
void imprime (int m[][COLUMNAS]);
void lee (int m[][COLUMNAS]);
int main()
{
int m[FILAS][ COLUMNAS] ;

inicializa(m);
lee(m);
imprime (m);
}
```

❑ OPERACIONES CON PUNTEROS

➘ ASIGNACION A PUNTERO

Mediante una sentencia de asignación, para darle:

- una dirección con el operador &, o
- el valor de otro puntero, del mismo tipo base

Por ejemplo:

```
#include <stdio.h>
int main()
{
 int var=27;
 int *pvar, *pv;

          /*Asignación de una dirección a un puntero*/

pvar=&var;    /*pvar apunta ahora a var*/

          /*Asignación de un puntero a otro*/

pv=pvar;  /*pv también apunta a var*/

printf("Dirección de var: %p  contenido de var: %d\n",&var, var);
printf("Dirección apuntada por pvar: %p  contenido: %d\n",pvar, *pvar);
printf("Dirección apuntada por pv: %p  contenido de var: %d\n",pv, *pv);
}
```

➘ ARITMÉTICA DE PUNTEROS

- Un puntero contiene una dirección de memoria (un número): podemos sumar o restar una cantidad a un puntero, haciendo que el puntero apunte a otra dirección de memoria.

- Todos los operadores aritméticos de suma y resta son aplicables a punteros (**+ , - , += , -= , ++,--**)

- Semántica: Sumar (restar) la cantidad de 1 a un puntero, hace que el puntero se incremente (decremente) no en 1, si no en una cantidad igual al número de bytes del tipo base; es decir, se hace que el puntero apunte al siguiente dato en memoria de ese tipo.

- Si se restan dos punteros que apuntan a dos posiciones de un array, se obtiene el número de elementos que hay almacenados entre ambos punteros.

Por ejemplo:

```
int main(int argc, char * argsv[])
{
char cvar[5]={'a','b','c','d','e'};  //Tamaño: 5 bytes
char * pcvar;

int  ivar[5]={1,2,3,4,5};            //Tamaño: 5x2== 10 bytes
int * pivar,pivar2;

pcvar=&cvar;
```

```
printf("Letra %c,dirección %p",*pcvar,pcvar); //'a',FF00
pcvar++;
printf("Letra %c,dirección %p",*pcvar,pcvar); //'b',FF01
pcvar+=3;
printf("Letra %c,dirección %p",*pcvar,pcvar); //'e',FF04
pivar=pivar2=&ivar;
printf("Numero %i,direccion %p",*pivar,pivar);// 1, FF07
pivar++;
printf("Numero %i,direccion %p",*pivar,pivar);// 2, FF09
pivar+=3;
printf("Numero %i,direccion %p",*pivar,pivar);// 5, FF0F
printf("El tamaño del array de caracteres es%i",pivar-pivar2);
}
```

➘ COMPARACION DE PUNTEROS

Mediante una expresión relacional. Deben realizarse solo cuando las direcciones estén próximas (si no, pueden producir resultados erróneos). Normalmente la comparación de punteros se realiza cuando apuntan a un objeto común (p ej., a distintas posiciones en un array).

```
p=&array[0];
```

```
q=&array[5];
if (p<q) printf("p apunta a una posición anterior a q\n");
```

Es usual comparar (con ==) un puntero con la constante NULL (definida en stdlib.h)

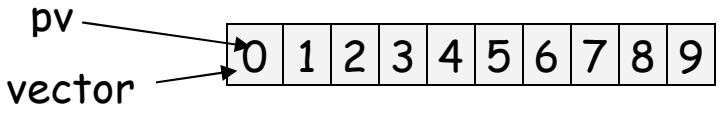
❑ PUNTEROS Y VECTORES

✓ Los operadores de punteros y vectores son intercambiables: se puede trabajar con los elementos de un array mediante operaciones de punteros y se pueden manipular punteros con el operador de indexación [] de arrays.
En realidad el identificador de un vector es un puntero a su primer elemento.

```
int *pv;
int vector[10];          //vector==&vector[0]

pv=vector;  // o pv=&vector[0]

if (pv == vector)  printf ("Esto siempre es cierto");
```



➘ <u>Desplazamiento sobre un array</u>

Los elementos del array se almacenan en posiciones consecutivas de memoria, así que para

referirse a un elemento del array basta con desplazar el puntero a la posición deseada :

```
vector       == &vector[0] == pv       == &pv[0]
(vector+1)   == &vector[1] == (pv +1)  == &pv[1]
(vector+2)   == &vector[2] == (pv+2)   == &pv[2]
```

➘ Modificación del contenido de una posición

Al tratarse de un puntero, al identificador de un vector se le puede aplicar el operador * para obtener el contenido de la dirección a la que apunta:

```
*vector     == vector[0]  ==  *pv     ==   pv[0]
*(vector+1) == vector[1]  ==  *(pv+1) ==   pv[1]
*(vector+1) == vector[2]  ==  *(pv+2) ==   pv[2]
```

En general *(vector +i)

➘ De igual manera para acceder a los elementos de una matriz m bidimensional con aritmética de punteros podemos usar:

((m + i) +j)

✓ Diferentes estilos para recorrer un array

1º. Operador de Indexación []

```
#include <stdio.h>
#define DIM 10
int main()
{
 int i, x=1, vector[DIM];

 for (i=0; i<DIM; i++)  //Escritura
       vector[i]=x;     //Indexación del vector con
[]

 for (i=0; i<DIM; i++) //Lectura
       printf("%d",vector[i]);
}
```

2º. Identificador del vector usado como puntero

```
#include <stdio.h>
#define DIM 10
int main()
{
 int i, x=3, vector[DIM];

 for (i=0;i<DIM; i++)  //Escritura
     *(vector+i)=x; //Identificador como puntero

 for (i=0;i<DIM; i++)  //Lectura
            printf("%d",*(vector+i));
}
```

3º. Desplazamiento del puntero cambiando su valor

```
#include <stdio.h>
#define DIM 10
int main()
{
 int  x=3, *p, vector[DIM];

for (p=&vector[0]; p<&vector[DIM]; p++)
    *p=x;

for (p=&vector[0]; p<&vector[DIM]; p++)
    printf("%d",*p);
}
```

❑ CADENAS DE CARACTERES

✓ En C una ***cadena*** se define como ***un array de caracteres*** de cualquier longitud que ***termina*** en un carácter nulo. El carácter nulo se especifica como ***'\0'***, cuyo valor ASCII es 0, es decir, el número es 0.

Para declarar un array de caracteres es necesario que sean de un carácter más que la cadena más larga que pueda contener.

```
char nif[10];  //10 ==8 dígitos+1 letra + '\0'
```

✓ Una constante literal cadena es una lista de caracteres encerrada en dobles comillas (El compilador se encarga de añadir el carácter nulo).

"hola" == | h | o | l | a | '\0' |

Una forma fácil de inicializar una cadena sería:

```
char nif[10]="12345678A";
char nif[10]=""//Cadena vacia:contiene carácter '\0'
```

✓ Recordar que en C los caracteres se representan mediante un byte en el que se almacena el código ASCII que codifica el carácter. Pero ese byte, también se puede interpretar como un número; por ejemplo '\n' es el 13, 'A' es 65 y 'a' es el 97. Por ello es válido escribir expresiones como 'b'-'a', que

da como resultado el número 1, ya que el código ASCII de la 'b' es el siguiente de la 'a'.

- ✓ La mejor forma de pasar una cadena a una función es declarar el parámetro formal como un puntero o como un vector sin especificar tamaño. El parámetro actual puede ser un puntero o un vector.

```
int función (char *cad)  //Alternativamente
cad[]
{   ……
}
int main ()
{
  char v[5]="hola";
  char * p;
  p=(char *) malloc(2 * sizeof(int));
  *p='a'; *(p+1)='\0';
  funcion(v);   funcion(p);
}
```

- ✓ Para que una función pueda devolver una cadena de caracteres se declara de tipo 'char *'

```
char *función (parámetros)
{
      return (...);
}
```

Para asignar el resultado de una función a una variable cadena, no podemos utilizar la asignación,

como hasta ahora hemos realizado. Tenemos que utilizar obligatoriamente la función ***strcpy***.

❑ FUNCIONES DE MANEJO DE CADENAS

✓ Estas funciones, que están en la librería '***string.h***' :

- No realizan reserva dinámica de memoria: Si una función necesita escribir una cadena en un parámetro, ese parámetro debe ser o un vector o un puntero a char inicializado con malloc o realloc.

- No comprueban los límites de los arrays, asi que es responsabilidad del programador asegurar que los arrays o punteros que se le pasan a estas funciones son lo suficientemente grandes

✓ Las funciones más utilizadas son las siguientes:

- **char * strcat(char * cadena1, char * cadena2)**

  Concatena (añade al final) una copia de cadena2 en cadena1 y añade al final de cadena1 el carácter nulo '\0'. El carácter nulo que originalmente tenía cadena1 se sustituye por el

primer carácter de cadena2. La cadena2 no se modifica en esta operación.
La función devuelve la (dirección de) cadena1 modificada por la concatenación.

Ejemplo: Programa que lee dos cadenas, pega la primera a la segunda y muestra el resultado por pantalla.

```
#include <stdio.h>
#include <string.h>
int main ()
{
      char cad1[30],cad2[30];
      gets(cad1);
      gets(cad2);
      strcat(cad2,cad1);
      puts(cad2);
}
```

- **int strcmp (const char*cad1, const char*cad2)**

Compara lexicográficamente, distinguiendo entre minúsculas y mayúsculas, dos cadenas que finalizan con el carácter nulo y devuelve un entero que se interpreta así:

Valor devuelto	Significado
0 < (negativo)	cad1 es menor que cad2
0	cad1 == cad2
> 0 (positivo)	cad1 es mayor que cad2

Las funciones stricmp y strcmpi: ignoran la diferencia entre minúsculas y mayúsculas.

Ejemplo: Programa que verifica la introducción de contraseñas. La palabra clave es "paso".

```
#include <stdio.h>
#include <string.h>
int claveOK();  //Función booleana
int main ()
{
   if (claveOK()){
      printf("Palabra clave correcta");
      return(1);
   }
   else{
     printf("palabra clave invalida\n");
     return(0);
   }
}
int claveOK()
{
    char s[30];
    int nIntentos=3;
    do{
     printf("Introduzca una palabra clave: ");
```

```
        gets(s);
        if (strcmp(s,"paso"))
            nIntentos--;   //Palabra clave no válida
        else
            break;          //Palabra clave válida
      } while(nIntentos>0);

      return(nIntentos);
   }
```

- **char *strcpy (char *cad1, const char *cad2)**

Copia el contenido de cad2 en cad1, devolviendo cad1. Es lo mas parecido que tiene C a la operación de asignación de una cadena a una variable.

Ejemplo: Programa que copia "hola" en 'cad'

```
#include <stdio.h>
#include <string.h>
int main ()
{
      char cad[80];
      strcpy(cad,"hola");
      puts(cad);
}
```

- **int strlen (const char *cad1)**

Devuelve el número de caracteres de una cadena sin contar el '\0' final.

- **char *strlwr (char *cad1)**

Convierte cad1 a minúculas.

- **char *strncat (char*cad1,const char*cad2,int n)**

Añade no más de n caracteres (un carácter nulo y los demás caracteres siguientes no son añadidos) de la cadena apuntada por cad2 al final de la cadena apuntada por cad1. El carácter inicial de cad2 sobrescribe el carácter nulo al final de cad1. El carácter nulo siempre es añadido al resultado.

Ejemplo: Programa que concatena dos cadenas e impide que se produzca desbordamiento

```
#include <stdio.h>
#include <string.h>
int  main ()
{ char s1[5], s2[5];
 int lon;
 gets(s1);   gets(s2);
 lon=4-strlen(s1);
 if(lon>=0)   strncat(s1,s2,lon);
 puts(s1);
}
```

- **char* strncpy(char *dest, const char*orig,int n)**

Copia no más de n caracteres (caracteres posteriores al carácter nulo no son copiados) de la cadena 'orig' a la cadena 'dest'.

Ejemplo: Programa que copia toda una cadena leída por teclado en otra asegurándose de que no se produce desbordamiento.

```
#include <stdio.h>
#include <string.h>
int main ()
{     char s1[30], s2[5];
      gets(s1);
      strncpy(s2,s1,4);
      puts(s2);
}
```

- **char *strupr (char* cad)**

Convierte 'cad' a mayúsculas

❑ FUNCIONES DE MANEJO DE CARACTERES

Estas funciones se aplican solo sobre caracteres y estan incluidas en las librerías '***ctype.h*** 'y '***stdlib.h***'. (atof, atoi, atol, itoa)

- ***double*** **atof(const char *cad)**

Convierte una cadena en un double. Devuelve el double correspondiente y 0 si la cadena no es un numero.

- ***int atoi(char *cad)***

  Convierte una cadena en un entero. Devuelve el entero correspondiente y 0 si la cadena no es un entero.

- ***long int atol(const char *cad)***

  Convierte una cadena en un entero largo. Devuelve el entero correspondiente y 0 si la cadena no es un entero.

- ***int isalpha(int car)***

  Devuelve un valor distinto de 0 (verdadero) si el carácter es una letra del alfabeto y 0 (falso) en otro caso.

- ***int isdigit(int car)***

  Devuelve un valor distinto de 0 (verdadero) si el carácter es un dígito y 0 (falso) en otro caso. 0

- ***int isalnum(int car)***

  Devuelve un valor distinto de 0 (verdadero) si el carácter es una letra o un dígito y 0 (falso) en otro caso. 0

- ***int islower(int car)***

  Devuelve un valor distinto de 0 (verdadero) si el carácter es una letra en minúscula y 0 (falso) en otro caso. 0

- ***int ispunct(int car)***

  Devuelve un valor distinto de 0 (verdadero) si el carácter es un símbolo (';','@, …) y 0 (falso) en otro caso.

- ***int isspace(int car)***

  Devuelve un valor distinto de 0 (verdadero) si el carácter es un espacio (' ',tabulación, intro,…) y 0 (falso) en cualquier otro caso.

- ***int isupper(int car)***

  Devuelve un valor distinto de 0 (verdadero) si el carácter es una letra en mayúsculas y 0 (falso) en cualquier otro caso.

- ***int itoa(int entero,char *cad,int base)***

  Convierte un entero a una cadena. La base especifica la base que debe ser usada en la conversión. Debe estar entre 2 y 36. Si el

entero es negativo y la base es 10 el primer carácter de la cadena será el signo menos (-).

- ***int tolower(int car)***

  Devuelve el equivalente en minúscula del carácter. Si el carácter no es una letra del alfabeto no sufre ningún cambio.

- ***int toupper(int car)***

  Devuelve el equivalente en mayúsculas del carácter. Si el carácter no es una letra del alfabeto no sufre ningún cambio.

❑ TIPOS DE DATOS DEFINIDOS POR EL USUARIO

El lenguaje C permite al programador crear tipos de datos propios mediante los siguientes recursos:

- **registro o estructura**: agrupación de varias variables (de igual o distinto tipo) bajo un mismo nombre

- **campo de bits**: acceso a memoria a nivel de bits. No los estudiaremos

- **unión**: permite que la misma porción de memoria sea compartida por dos o más variables de diferente tipo.

- **enum**: creación de un tipo enumerado.

- **typedef**: nuevo nombre para un tipo ya existente

❑ REGISTROS O ESTRUCTURAS

✓ Concepto

Registro: Objeto que permite agrupar varias variables de igual o distinto tipo bajo un mismo nombre.

Las variables que forman el registro se llaman ***campos*** del registro.

Un registro es una especie de '"ficha" referente a un tema concreto y dividida en distintos apartados.

Ejemplo: Registro de datos personales de un individuo

✓ Declaración

1º. De un *tipo* de registro

```
struct nombreTipoRegistro {
tipo campo1;
tipo campo2;
…
tipo campoUltimo;
};
```

Para declarar una variable de tipo registro

```
struct nombreTipoRegistro   variable;
```

Ejemplo

```
struct datosP {
            char nombre[30];
            char direccion[40];
            float talla;
            unsigned long int codigoPostal;
          };

struct datosP persona;
```

Podria emplearse el mismo nombre:

```
                          //Variable 'datosP'
struct datosP datosP;   //de tipo 'struct datosP'
```

2º. De *varias* ***variables*** de tipo registro

```
struct nombreTipoRegistro{
tipo campo1;
…
tipo campoUltimo;
} variable1,variable2, …, variableN;
```

Ejemplo

```
struct datosP {
        char nombre[30];
        char direccion[40];
        float talla;
        unsigned long int codigoPostal;
} persona1, persona2, persona3;
```

3º. De *una* ***variable*** de tipo registro

```
struct{
tipo campo1;
…
tipo campoUltimo;
} variable1;
```

Ejemplo

```
struct {
        char nombre[30];
        char direccion[40];
        float talla;
```

```
        unsigned long int codigoPostal;
    } persona;
```

✓ Acceso

1º. Operador '.' (punto)

Para acceder a un campo de una variable de tipo registro se aplica el operador punto '.':

variableRegistro.variableCampo

Se utiliza exactamente igual a como se utilizaría una variable del mismo tipo que ese campo.

Ejemplo

```
Persona1.codigoPostal=11010;
strcpy(persona1.nombre,"Alonso Lucas Pi");
```

2º. Operador '=' (asignación)

El operador de asignación está definido para el tipo registro; es decir se puede asignar a una variable de un tipo registro el valor de otra variable registro *del mismo tipo registro*:

Ejemplo

```
persona2=persona1;
```

La variable persona2 contiene en cada uno de sus campos los mismos valores que hay en cada uno de los campos de la variable persona1.

3º. Operador '->'

Si una variable es de tipo "puntero a registro", se puede acceder al valor de un campo del registro apuntado por ese puntero con el operador '->'.

Ejemplo

```
struct datosP p, *pr;

p.talla=1.79;
pr=&p;
printf("La talla es %f", pr->talla);//Mostrará 1.79
```

Ejemplo

```
#include<stdlib.h>
#include<string.h>

struct empleado{
char nombre[40];
char sexo;
float sueldo;
};
int main()
{
 struct empleado e1,e2, *pe;

 printf("¿Datos empleado?: Nombre, sexo(h/m), sueldo");
 scanf("%s ,%c, %f",e1.nombre,&e1.sexo,&e1.sueldo);

 e2=e1;

 printf("¿Nombre del segundo empleado?");
 scanf("%s",e2.nombre);

 if(!strcmp(e1.nombre,e2.nombre))
   printf("El nombre de ambos empleados es el mismo");

 pe=&e2;

 if (pe->sexo == e1.sexo)
```

printf("Siempre se mostrará: Los sexos son iguales");

printf ("El sueldo doblado: %f",**pe->sueldo** + **e1.sueldo**);
}

✓ Uso de registros

- VECTORES DE REGISTROS

Es frecuente que se necesite mantener una lista de registros (por ejemplo la lista de registos de empleados de una empresa). La manera mas simple de implementar esa lista es mediante un vector, del tamaño adecuado, y cuyo tipo base es de tipo registro.

Ejemplo

struct empleado Empleados[100];

Para acceder a cada uno de los campos de un registro del vector hay que indicar:
–posición del vector donde se encuentra el registro (**v[i]**)
–nombre del campo (.**campo**)

Ejemplo

printf("Sueldo 4º empleado: %f",
Empleados[3].sueldo);

Ejemplo

```
#include <stdio.h>
#include<stdlib.h>
#include<string.h>
#define MAXEMP 100
struct empleado{
                    char nombre[40];
                    char sexo;
                    float sueldo;};
int main()
{
 struct empleado Empleados[MAXEMP];
 int i,salir=0,nEmpleados;
 for(i=0; i<MAXEMP && !salir ;i++){
   fflush(stdin);
   printf("Nombre empleado %i",i+1);
   gets(Empleados[i].nombre);
   if(!strcmp(Empleados[i].nombre,"")    salir=1;
   else{
        printf("sexo(h/m), sueldo");
        scanf( "%c, %f",
        &Empleados[i].sexo,
        &Empleados[i].sueldo); }
 }
 nEmpleados=(i==MAXEMP?: i : i-1);
 for(i=0;i<nEmpleados;i++)
     Empleados[i].sueldo*=1.1;

for(i=0;i<nEmpleados;i++)
  printf("El sueldo incrementado un 10 % de %s es
%f",
```

Empleados[i].nombre,
Empleados[i].sueldo);}

- REGISTROS CON CAMPOS DE TIPO VECTOR

Un campo de un registro puede ser un vector o una matriz. Se accede primero al campo (operador '.') y luego se indexa (operador '[]').

Por ejemplo

```
struct quiniela{
  char resultado[15];  //'1', 'x', '2'
  char partido [15][40]; /*Lista de 15
partidos*/
  unsigned char jornadaNumero;
} q;
```

printf("Partido pleno al 15 es %s , resultado %c",
q.partido[14], q.resultado[14]);

Se puede definir un vector de tipo registro, el cual tenga campos de tipo vector.

Por ejemplo

struct quiniela liga[NPARTIDOS];

printf("Partidos de pleno al 15 de la liga 2004/05");

```
for(i=0;i<NPARTIDOS;i++)
  printf("Jornada %i, Partido %s , resultado %c",
i+1,liga[i].partido[14],liga[i].resultado[14]);
```

- REGISTROS CON CAMPOS DE TIPO REGISTRO

Un registro puede tener campos que sean de tipo registro. Se accede primero al campo de tipo registro (operador '.') y luego se accede a alguno de los campos de ese registro (operador '.')

Ejemplo

```
struct direccion{
                    char calle[40];
                    unsigned char numero;
                    char cp[6];
};
struct empleado {
                      char nombre[40];
                      struct direccion direccion;
                      long int salario;
} empleado;

strcpy(empleado.nombre,"Pepe Pérez Pi");
empleado.salario=1500;
strcpy(empleado.direccion.cp,"11010");
empleado.direccion.numero=5;
strcpy(empleado.direccion.calle,"Real");
```

```
printf("Codigo postal del empleado %s es %s",
        empledo.nombre,
        empleado.direccion.cp;)
```

✓ Registros y Funciones

- **Paso de parámetros**

  – El tipo registro debe ser global (fuera de cualquier función)

  – En la cabecera de la función se especifica el tipo de registro y el nombre del parámetro formal

  – En la invocación de la función se escribe el nombre del parámetro actual

Ejemplo

```
struct Treg {int n; char c[100]};

void sumaR(struct Treg r1, struct Treg r2,
                struct Treg  * rs)
{
 rs->n=r1.n+r2.n;
 strcpy(rs->c,r1->c);
 strcat(rs->c,r2->c);
}
```

```
int main()
{
   struct Treg ra, rb,r;
        .......
   sumaR(ra,rb,&r);
}
```

- **Devolución de un registro**

Se puede definir una función de tipo registro. Si el registro es muy grande (ocupa muchos bytes) es aconsejable pasarlo como parámetro simulando el paso por referencia y trabajar con él a través de un puntero. Si no se hace así, cada vez que se devuelva un registro (return) se debe hacer una copia de todos los bytes del registro y el programa es mas lento.

Ejemplo

```
struct empleado {
                        char nombre[40];
                        struct direccion direccion;
                       long int salario;}
};
```

```
struct empleado leeReg()
{
 struct empleado re;

 scanf("…", &re.salario…);
 return (re);
}
int main()
{
  struct empleado r;
  r=leeReg();
}
```

```
void leeReg(struct empleado
*re)
{

 scanf("…",re->salario);

}

int main()
{
  struct empleado r;
  leeReg(&r);
}
```

❑ UNIONES

Una unión es un registro en el que todos los campos se almacenan en la misma región de memoria (se reserva memoria para el campo de mayor tamaño).

La **declaración** de uniones y registros es similar:

union *TUnion* **{** tipo campo1; tipo campo2; … tipo campoUlt; };	**union** *TUnion* **{** tipo campo1; tipo campo2; … tipo campoUlt; } v1,v2,…,vN;	**union {** tipo campo1; tipo campo2; … tipo campoUlt; }v1;

El **acceso** también es similar (operadores '.','->','=')

Una unión reserva espacio suficiente para almacenar el tipo de datos más grande que la compone; de esta forma todos los elementos que la forman tienen sitio suficiente para almacenar valores en ella, siempre y cuando lo hagan de uno en uno. Es decir, una unión no permite que haya dos elementos con valores válidos al mismo tiempo.

Ejemplo

```
union { int n; char c;} varUnion;
```

Tamaño: 2 bytes

```
varUnion.n=9999;
varUnion.c='A';
```

varUnion.c == 'A', pero varUnion.n!=9999 pues el primer byte ha cambiado (contiene el código ASCII de la 'A').

❑ ENUMERACIONES

✓ Concepto

Definir un tipo enumerado consiste en indicar explícitamete los valores que puede tomar una variable de ese tipo.

En C una enumeración es un conjunto de constantes enteras con nombre que especifica todos los valores del tipo.

✓ Declaración

enum Tenumerado **{**E1, E2, …, En**}** lista_de_variables;

o

enum Tenumerado **{**E1, E2, …, En**}**;

- Cada elemento Ei es un identificador cuyo valor es un número entero.

- Por defecto el valor del primer elemento es 0 y el valor del siguiente es el valor del anterior +1

- Se puede asignar un valor entero concreto a un elemento escribiendo 'Ei=cteLiteralEntera'

✓ Uso

Ejemplo

```
enum moneda {
        centimo=1, doscentimos,
        cincocentimos=5,
        diezcentimos=10,
        cincuentacentimos=50,
        euro=100, doseuros=200
};

//Variables 'dinero1' y 'dinero2' de tipo 'enum moneda'
enum moneda dinero1,dinero2

dinero1=euro;        //valor entero 'euro' con valor 100
dinero2=doscentimos; //valor 'doscentimos' es 2

  printf("El valor de un euro es %d centimos",
                        dinero1);
 printf("y de dos centimos es %d centimos", dinero2);
```

❑ TYPEDEF

✓ Concepto

Permite crear alias de tipos ya definidos; es decir se puede definir otro nombre para un tipo ya existente. La razón para hacer esto es conseguir que el programa sea más legible.

✓ Declaración

- **typedef tipoAntiguo aliasNuevo:**

Ejemplo

```
typedef int entero;
typedef float real;
typedef struct empleado empleado;

entero edad;
real sueldo;
empleado Empleados[100];
```

- **typedef struct{. . .} tipoEstructura;**

  Permite darle un nombre corto a una larga definicion de estructura o registro.

  Ejemplo

```
typedef struct{
                    char calle[40];
                    unsigned char numero;
                    char cp[6];
 } direccion;

 typedef struct {
                char nombre[40];
                direccion direccion;
                long int salario;
  } empleado;

  empleado Empleados[100];
```

- **typedef tipobase tipoVector[DIM]**

  Evitar tener que escribir en todas las declaraciones de variables los parámetros de la(s) dimensión(es)

  Ejemplo

  typedef char cadena[DIM];

  /*la variable 'cad1' es un vector de caracteres de dimensión DIM*/

  cadena cad1;

  typedef int notas[ALUM][ASIG];

  /*Las variables 'eso4A' y 'eso4B' son matrices de ALUMxASIG enteros*/

  notas eso4A,eso4B;

  /*En la siguiente declaración de función se hace evidente la claridad obtenida al usar los alias */

```
void imprimirNotas (cadena tutor, notas
curso);
```

❑ FICHEROS

✓ Para el lenguaje C un flujo de datos es una corriente o flujo de bytes que puede hacerse corresponder con un fichero físico en disco.

✓ Para utilizar ficheros en primer lugar se debe declarar un flujo de datos como FILE * siendo el tipo FILE una estructura definida en la cabecera <stdio.h> (así como el resto de funciones para manejar ficheros).

✓ Los ficheros pueden ser:

- Ficheros de Texto: si están divididos en líneas que contienen caracteres y que acaban en un salto de línea.
- Ficheros binarios: Si tienen cualquier otra estructura.

✓ Antes de trabajar con un fichero debemos abrirlo. Esta operación conecta el flujo con el fichero físico en disco. A partir de ese momento trabajaremos con el fichero a través del flujo con el que está conectado. En la operación de apertura es dónde se decide de qué tipo será el fichero (binario o texto).

- ✓ Una vez que se acabe de trabajar con un fichero este debe cerrarse, desconectando así el flujo del fichero físico.

✓ Declaración de un flujo

FILE * id_var_fichero;

✓ Apertura y cierre de un fichero

Antes de poder leer o escribir datos en un fichero hay que abrirlo mediante la función fopen() definida del siguiente modo:

FILE * fopen (const char *nombre_fichero, const char *modo_apertura)

Como parámetros recibe dos cadenas:

Nombre_fichero: es el nombre o ruta de acceso del fichero físico que deseamos abrir.

Modo_apertura: especifica el modo de apertura, indica el tipo de fichero (texto o binario) y el uso que se va a hacer de él lectura, escritura, añadir datos al final, etc.

Los modos disponibles son:

Modo	Descripción
r	abre un fichero para lectura. Si el fichero no existe devuelve error.
w	abre un fichero para escritura. Si el fichero no existe se crea, si el fichero existe se destruye y se crea uno nuevo.
a	abre un fichero para añadir datos al final del mismo. Si no existe se crea.
+	símbolo utilizado para abrir el fichero para lectura y escritura.
b	el fichero es de tipo binario.
t	el fichero es de tipo texto. Si no se pone ni b ni t el fichero es de texto.

Los modos anteriores se combinan para conseguir abrir el fichero en el modo adecuado.

Por ejemplo, para abrir un fichero binario ya existente para lectura y escritura el modo será "rb+ "; si el fichero no existe, o aun existiendo se desea crear, el modo será " wb+". Si deseamos añadir datos al final de un fichero de texto bastará con poner "a", etc.

La función fopen() devuelve un flujo que usaremos para trabajar con el fichero, a través del resto de funciones de manejo de ficheros, o NULL en caso de que el fichero no se haya podido abrir.

Ejemplo

Abre un fichero de texto existente para lectura

```
#include <stdio.h>
#include <stdlib.h>
…
FILE *pf
…
if ((pf= fopen("prueba.txt", "r"))==NULL)
{
printf("Error de apertura\n");
exit(1);
}
```

Una vez que se ha terminado de utilizar el fichero debemos cerrarlo con la siguiente función:

int fclose(FILE* id_var_fichero);

Devuelve 0 si el cierre se ha realizado con éxito y la macro EOF en caso contrario.

✓ Funciones de lectura

- Lectura de caracteres de un fichero:

int fgetc (FILE *f);

Devuelve el carácter del fichero situado en la posición actual o EOF si es final del fichero. Avanza el indicador a la siguiente posición.

- Lectura de cadenas de un fichero:

char * fgets(char * s, int tam, FILE * f);

Lee una cadena de caracteres del fichero y la copia en la cadena apuntada por s, el tamaño de la cadena leida vendrá determinado por el entero tam, por el carácter fin de línea o por el carácter de fin de fichero EOF. Devuelve un puntero a la misma cadena.

- Lectura formateada de un fichero:

int fscanf(FILE * f, const char * formato, …);

Funciona exactamente igual que scanf pero leyendo del fichero en vez de de la entrada estándar (flujo stdin).

- Lectura de bloques de un fichero:

size_t fread (void * p, size_t tam, size_t n, FILE * f);

Lee tantos datos como indique **n** del fichero, colocando los datos leídos a partir de la dirección **p**. Los datos tienen que tener tantos bytes como especifique **tam**. La función fread devuelve el número de elementos leídos, y el valor devuelto debe coincidir con **n**.

✓ Funciones de escritura

- Escritura de caracteres en un fichero:

int fputc (int c, FILE *f);

Escribe el carácter c en la posición actual del fichero. Devuelve el carácter escrito o EOF en caso de error. Avanza el indicador a la siguiente posición.

- Escritura de cadenas en un fichero:

int fputs(const char * s, FILE * f);

Escribe la cadena de caracteres apuntada por s en el fichero. Devuelve EOF si se produce un error.

- Escritura formateada en un fichero:

int fprintf(FILE * f, const char * formato, …);

Funciona exactamente igual que printf pero escribiendo en el fichero en vez de de la salida estándar (flujo stdout).

- Escritura de bloques en un fichero:

size_t fwrite (const void * p, size_t tam, size_t n, FILE * f);

Escribe tantos datos como indique **n** el fichero, tomando los datos a partir de **p**. Cada dato leido tiene que tener tantos bytes como especifique **tam**. La función **fwrite** devuelve el número de elementos escritos, este valor debe coincidir con **n**.

Las funciones de lectura y escritura de bloques suelen usarse con ficheros binarios, para permitir leer o escribir datos de cualquier tipo: números, caracteres, o incluso estructuras completas: es decir que nos permitan leer o escribir zonas de memoria que puedan contener cualquier tipo de datos. Son funciones muy potentes, con una sola llamada podemos leer o escribir un vector, una estructura o incluso un vector de estructuras.

Ejemplo 1:

```
FlLE *f;
int v[6], elem_escritos, num;
f = fopen (“datos.cht ", "wb ");
/* Para escribir los 3 últimos elementos de v (el 2, el 3 y el 4) */
elem-escritos = fwrite (&v[2], sizeof(int), 3, f );
/* Para escribir el primer elemento de v, valen las 2 instrucciones
siguientes */
fwrite (v, sizeof (int), 1, f );
fwrite (&v[0], sizeof(int), 1, f );
/* Para escribir un entero valen las dos siguientes */
fwrite (&num, sizeof(int), 1, f);
fwrite (&num, sizeof(num), 1, f);
```

Ejemplo 2:

```
f = fopen (“datos.dat ", "rb ");
elem-escritos = fread (&v[2], sizeof(int), 3, f);
fread (v, sizeof(int), 1, f);
fread (&V[0], sizeof(int), 1, f);
fread (&num, sizeof(int), 1, f);
fread (&num, sizeof(num), 1, f);
```

✓ Funciones de posicionamiento

- Conocer la posición del indicador:

long int ftell (FILE *f);

Para ficheros binarios devuelve la posición del indicador en número de bytes medida desde el principio.

- Posicionamiento directo:

int fseek (FILE * f, long int adonde, int desdedonde);

Posiciona el indicador en la posición **adonde**, medida en bytes desde donde indique el argumento **desdedonde**. Teniendo en cuenta las siguientes constantes de desplazamiento:

SEEK_SET desde el principio
SEEK_CUR desde donde estamos
SEEL_END desde el final

- Rebobinamiento:

void rewind (FILE *f);

Posiciona el indicador al principio del fichero.

❑ ASIGNACIÓN DINÁMICA DE MEMORIA

✓ Los vectores se declaran con un cierto tamaño (número de casillas) máximo:

- Durante una ejecución del programa es posible que algunas posiciones nunca se utilicen: se reserva una parte de la memoria que no se usa
- Por el contrario, puede que el tamaño que estimó resulte demasiado pequeño y falten posiciones

✓ La gran ventaja de trabajar con **punteros** es que podemos utilizarlos como arrays dinámicos, es decir, no se declara su longitud a priori en tiempo de compilación, sino que se decide el tamaño que tendrán en tiempo de ejecución. A esto se le llama **reserva o asignación dinámica de memoria**.

Se puede realizar la reserva dinámica de memoria siguiendo dos enfoques:

1. Averiguar en tiempo de ejecución el número de casillas que necesitan y reservar (en tiempo de ejecución) esa cantidad de memoria.

2. Ir reservando casillas conforme vayan haciendo falta durante la ejecución del programa.

➘ RESERVA DINÁMICA DE UN BLOQUE DE MEMORIA

```
#include <stdio.h>
#include <stdlib.h>/*Librería que incluye las funciones de manejo de memoria*/
int main()
{
int i, x=1, nElementos, *vector;
/*'vector' se declara como un puntero pero su dimensión aún no se conoce*/

printf("Numero de elementos: ");
scanf("%d",&nElementos);

 /*En el puntero a entero 'vector' se almacena la dirección de memoria dada por 'malloc', que serán donde comience el vector*/

if((vector=(int*)malloc(nElementos*sizeof(int)))!=NULL)
//Se comprueba si se asignó  memoria a 'vector'
{      for (i=0;i<nElementos;i++)
            *(vector+i)=x;  //ponemos todas las casillas
                                a 1
       for (i=0;i<nElementos;i++)
            printf("%d",*(vector+i));
```

```
    }
else
   printf("Error. No se ha reservado memoria");
}
```

La función **malloc** se utiliza para reservar memoria para una determinada variable. Dicha variable debe ser un puntero, y el formato de uso de malloc es el siguiente:

puntero = (tipobase*) malloc (nº_elementos * sizeof(tipobase))

- Como argumento 'malloc()' recibe un número entero positivo que indica la cantidad de memoria que se desea reservar. Lo usual es calcular esa cantidad de memoria en función del tipo de dato que se quiere almacenar y la cantidad de datos de ese tipo que se almacenarán. sizeof reserva exactamente la memoria necesaria para cada tipo de datos, por ejemplo, en el programa anterior reservaría para cada entero 2 bytes de memoria (esta cantidad puede variar de un tipo de ordenador a otro, por eso hay que utilizar 'sizeof()' y no especificar un número); dicha cantidad de memoria se multiplica por el número de casillas que queremos tener en el vector dinámico

- Como resultado 'malloc()' devuelve un puntero a void (void *), que hay que convertir (conversión explícita) al tipo "puntero al tipo base" (int * en el

ejemplo) y guardar dicha dirección en una variable puntero de ese mismo tipo. Dicho de otra manera, se almacena en puntero la dirección de la primera casilla del vector dinámico. Esa variable puntero es la que nos permite acceder a la memoria que se acaba de reservar.

- Nótese que en un puntero guardamos simplemente la dirección de la primera posición y a partir de ahí accedemos al resto de posiciones; ***es responsabilidad del programador asegurar que en el programa nunca se accederá mas allá de la última posición reservada***. En el ejemplo anterior, si en el bucle for la condición es i<=nElementos, el programa está accediendo a una posición:

  (vector[nElementos]==*(vector+nElementos))

  que está fuera del bloque de memoria que se ha reservado.

- Si no se ha realizado correctamente la asignación de memoria, malloc devuelve el puntero nulo NULL (constante definida en stdlib.h, cuyo valor suele ser 0), por lo que ***siempre, después de malloc se debe comprobar su resultado*** por si no se ha hecho la reserva

↘ REASIGNACIÓN DINÁMICA DE UN BLOQUE DE MEMORIA

El segundo enfoque de manejo de memoria dinámica consiste en ir reservando casillas o eliminándolas conforme vaya haciendo falta utilizando la función **realloc**.

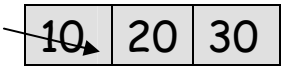
puntero = (tipobase*) realloc (puntero,
nº_elementos * sizeof (tipobase))

realloc libera la región de memoria ocupada por 'puntero' y le asigna otra nueva, en la que copia todo lo que habia en la región anterior (y quepa en la nueva).

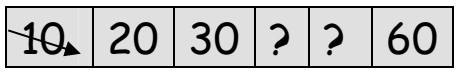
El inconveniente es que cada invocación a realloc (o malloc) conlleva una cierta cantidad tiempo, asi que ***no es una buena política solicitar con mucha frecuencia pequeñas porciones de memoria***

Por ejemplo,

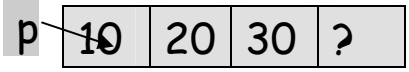
```
p=(int *) malloc(sizeof(int)*3);
p[0]=10; p[1]=20; p[2]=30;
```




```
p=(int  *) realloc (p,6*sizeof(int));
p[5]=60;
p
```



```
p=(int  *) realloc (p,4*sizeof(int));
```



Ejemplo de uso de realloc:

```
#include <stdio.h>
#include <stdlib.h> /*librería que incluye librerías de
                              manejo de memoria*/

int main()
{
int i=0, j, x=1, n, *vector;
vector=NULL;

do{
  if (vector==NULL)
      vector=(int *) malloc (sizeof(int));
 else
     vector=(int *) realloc (vector,(i+1)*sizeof(int));
       //reserva una casilla mas para el vector
     vector[i]=x;
     i++;
 }while (i<5);

for (j=0;j<i;j++)
       printf("%d", vector[j]);

free(vector);
}
```

↘ LIBERACIÓN DE UN BLOQUE DE MEMORIA

Es conveniente liberar la memoria dinámica que el programa haya solicitado y ya no necesite, para evitar que se agote la memoria dinámica.

- **free(p)**

  - Donde p es un puntero válido, es decir, un puntero que contenga una dirección obtenida previamente mediante una invocación a 'malloc()' o 'realloc()'.

  - Si p no es un puntero válido, a partir de ese momento la gestión dinámica de memoria es imprevisible. Nuevamente es responsabilidad del programador procurar que esto no ocurra nunca.

  Por ejemplo, free(vector) libera las posiciones de memoria asignadas a vector, que podran ser utilizadas otra vez por el programa.

- **realloc(vector,0)**

  libera toda la memoria ocupada por el vector sin reservar ninguna, por lo que es equivalente a free(p).

❑ MATRICES DINÁMICAS

Cuando se quiere trabajar con matrices bidimensionales cuyo tamaño es desconocido en tiempo de compilación lo que se hace es crear dinámicamente un vector de punteros a las filas, que a su vez se crean dinámicamente. La estructura completa queda apuntada por un puntero a puntero. Este método tiene la ventaja de permitir trabajar sobre la matriz con la notación usual de índice.

Ejemplo de reserva de una matriz dinámica:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int i, nFilas,nColumnas, **matriz;
  printf("Numero de Filas: ");
  scanf("%d",&nFilas);
  printf("Numero de Columnas: ");
  scanf("%d",&nColumnas);
if((matriz=(int**)malloc(nFilas*sizeof(int*)))==NULL)
    printf("Error. No se ha reservado memoria para el
    vector de punteros a las filas");
else
  for (i=0;i<nFilas;i++)
     if((matriz[i]=(int*)malloc(nColumnas
       *sizeof(int))) == NULL)
            printf("Error. No se ha reservado memoria
                para las filas");
}
```