Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga

Programación en C: Conceptos Avanzados

Programación de Sistemas para Control de Procesos
www.lcc.uma.es/~pscp
(Ingeniería Técnica Industrial en Electrónica)

José Galindo

Índice de contenidos

1. **Operador `sizeof`.**
2. **Preprocesador :**
   - **Constantes simbólicas.**
   - **Macros**
3. **Arrays :**
   - **Conceptos Básicos.**
   - **Arrays y Punteros.**
4. **Memoria Dinámica:**
   - **Arrays Dinámicos.**
   - **Arrays de Punteros.**
5. **Tipos Enumerados.**
6. **Clases de Almacenamiento de Variables.**
7. **Operadores a Nivel de Bits.**
8. **Campos de Bits.**
9. **Uniones.**
10. **Interrupciones Software.**
11. **E/S por Ficheros.**
12. **Tipos Abstractos de Datos (TAD): Introducción.**
13. **Estructuras de Datos Dinámicas.**

Para el seguimiento de estos contenidos se supone que se conocen los conceptos más básicos de la programación en **Lenguaje C:** funciones, arrays, estructuras, paso de argumentos por valor y por referencia...

Operador `sizeof`

- Este **operador** determina el **Tamaño en Bytes** de su operando:
  - **Si su operando es un tipo** de dato debe ponerse entre paréntesis.
    - **Ejemplo:** `sizeof (int)` es el tamaño de cualquier variable de tipo `int`.
  - **Si el operando no es un tipo**, sino una expresión, los paréntesis pueden quitarse, aunque también pueden ponerse (ya que los paréntesis pueden formar parte de cualquier expresión).
    - **Ejemplo:** `sizeof x` devuelve los bytes de memoria que requiere la variable **x**.
  - Al calcular el tamaño de una expresión esa expresión **no** es ejecutada (o evaluada).
    - **Ejemplo:** Al utilizar, `sizeof (a=b+1)` no asigna ningún valor a la variable **a**.
  - **Si el operando es un array** se devuelve el tamaño, en bytes, de todo el array.
    - Por tanto, es fácil descubrir el **número de elementos del array** dividiendo ese valor por el tamaño de cada uno de sus elementos.
    - Es una pena, pero esa técnica no funciona si se usa en el argumento de una función, ya que dicho argumento es sólo un puntero y `sizeof` devuelve el tamaño del puntero.

Constantes simbólicas: `#define`

- **Preprocesador :** Procesa el programa fuente ANTES de su compilación.
  - Sus órdenes empiezan por el símbolo **#** (sostenido, almohadilla).
- **`#define`:** Permite definir constantes de forma que podrán usarse a lo largo del programa. Formato: **`#define <nombre> <valor>`**
  - El preprocesador sustituye el `<nombre>` por el `<valor>`, en todas las ocasiones en las que aparezca. La sustitución se hace de forma automática, de forma que pueden generarse errores en la fase de compilación.
  - El `<nombre>` suele ponerse siempre todo en mayúsculas.
  - Ejemplos y usos:

```
#define LOG5  0.698970004
#define LN5   1,609437912
#define DOBLELN5 LN5*2
#define RESULTADO "\n- El resultado es el siguiente: "
#define FIN printf("\n- FIN.")
...
printf(RESULTADO);
printf("\n Logaritmo decimal de 5: %f", LOG5);
printf("\n Su mitad: %f", LOG5/2);
printf("\n Su cuadrado: %f", LOG5*LOG5);
printf("\n Doble del neperiano de 5: %f", DOBLELN5);
FIN;
```

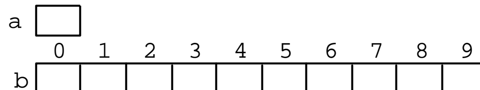
Macros: `#define`

- **`#define:`** Permite definir órdenes, usualmente simples, que simplifican la construcción de programas y los hacen más rápidos que usando funciones. Pueden tener argumentos que se ponen entre paréntesis.
  - Ejemplo: `#define DOBLE(X) 2*X`
  - Así, la expresión `DOBLE(5)` se sustituirá en el preprocesamiento por `2*5`.
  - Para evitar **errores** cuando la macro se usa en expresiones más complejas, se debe poner entre paréntesis cada argumento y la expresión completa.
  - Ejemplo: `#define SUM_CUAD(X,Y) X*X + Y*Y`
    - **Genera errores en expresiones como la siguiente:**
      `SUM_CUAD(a-b,2)` **que se traduce como** `a-b*a-b + 2*2`
      y se interpreta como: `a-(b*a)-b + (2*2)`
    - **Solución:** `#define SUM_CUAD(X,Y) (X)*(X) + (Y)*(Y)`
      - **La expresión ahora se traduce bien como:** `(a-b)*(a-b) + (2)*(2)`
    - **También genera errores en expresiones como la siguiente:**
      `z/SUM_CUAD(2,3)` **que se traduce como** `z/(2)*(2) + (3)*(3)`
      y se interpreta como: `((z/2)*2) + 3*3`
    - **Solución:** `#define SUM_CUAD(X,Y) ((X)*(X) + (Y)*(Y))`
      - **La expresión ahora se traduce bien como:** `z/((2)*(2) + (3)*(3))`
  - Este tipo de macroinstrucciones también pueden implementarse usando funciones, pero las macros son independientes del tipo de dato.
    - **Las macros generan programas más grandes pero más rápidos.**

Arrays: Conceptos Básicos

- **Consideremos las siguientes declaraciones:**

```
int a;
int b[10];
```



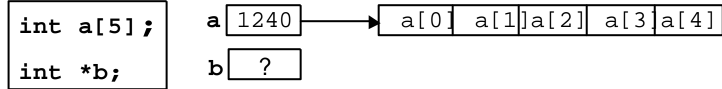
  - El tipo del valor de la variable `a` es un entero.
  - Cada elemento del array **`b`** `(b[0], ...,b[9])` es un entero y puede ser usado en cualquier contexto donde un entero pueda usarse.
  - Pueden accederse utilizando una variable como índice (dentro de un bucle, por ejemplo): **`b[i]`**
- **¿De qué tipo es `b`?**
  - NO se refiere al array completo.
  - El valor del nombre de un array es un "**Puntero Constante**" que tiene la dirección del primer elemento del array.
  - Por tanto, **`b`** es un puntero constante a un entero.
  - EXCEPCIÓN: **`sizeof(b)`** devuelve el tamaño, en bytes, del array y no el de un puntero. Pero si **`b`** fuera el argumento de una función entonces devolverá el tamaño del puntero (no del array).

Arrays y Punteros

- Hay una fuerte relación entre **arrays y punteros**, pero los **arrays** y los **punteros** NO son equivalentes:

```
int a[5];

int *b;
```



- **<u>DIFERENCIAS</u>:**
  - Declarar un array reserva memoria para **TODOS** los elementos.
  - Declarar un puntero reserva memoria **SÓLO** para el puntero.
  - El nombre de un array almacena una dirección fija (**constante**) que apunta al principio de este espacio. NO se puede cambiar el valor de la **constante**.
    - **El nombre de un array es la dirección de memoria del primer elemento: `a` º `&a[0]`**
  - Una variable puntero almacena una dirección de memoria que puede ser modificada. La variable puntero NO está inicializada para apuntar a ningún espacio existente, por lo que inicialmente tiene cualquier valor no válido. Para indicar que un puntero no apunta a ningún sitio se le puede asignar el valor **NULL**.

Arrays y Punteros

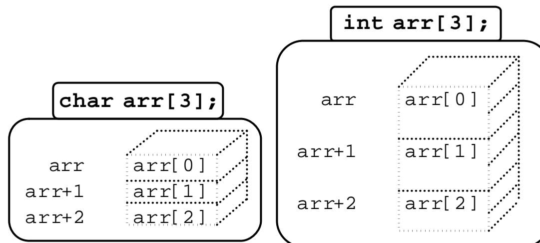
- Los arrays y los punteros usan diferentes notaciones de indexación.
- Así, las siguientes expresiones son **equivalentes:** `float arr[5];`

Usando índices		Usando punteros
`arr[0]`		`*(arr)`
`arr[1]`		`*(arr+1)`
`arr[2]`		`*(arr+2)`
`arr[3]`		`*(arr+3)`
`arr[4]`		`*(arr+4)`
`arr[índice]`	º	**`*(arr+índice)`**

<u>Direcciones</u>

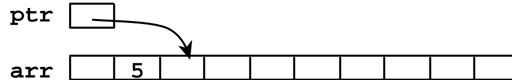
arr
arr+1
arr+2
arr+3
arr+4

El número de posiciones que se incrementa un puntero depende del tipo de datos del array. Suponemos que un `int` ocupa 2 bytes y un `char` 1 byte.



Arrays y Punteros

Ejemplos	Descripción
`int arr[10];`	Declara el array **`arr`** con 10 elementos.
`int *ptr;`	Declara un puntero a entero.
`arr[1]=5;`	Asigna 5 al segundo elemento del array **`arr`**.
`*(arr+1)=5;`	Equivalente a la instrucción anterior.
`ptr=&arr[2];`	Asigna a **`ptr`** la dirección del tercer elemento:



`ptr`	Equivale a **`arr+2`** y a **`&arr[2]`**
`*ptr`	Equivale a **`arr[2]`** y a **`*(arr+2)`**
`ptr[0]`	Equivale a **`arr[2]`**
`ptr + 6`	Equivale a **`arr+8`** o **`&arr[8]`**
`*ptr + 6`	Equivale a **`arr[2] + 6`**
`*(ptr + 6)`	Equivale a **`arr[8]`**
`ptr[-1]`	Equivale a **`arr[1]`**
`ptr[9]`	Equivale a **`arr[11]`** : ¡Fuera del array **`arr`**!

Acceder fuera de las posiciones de un array es un **fallo muy grave** del que no avisa el compilador. Es responsabilidad del programador usar los arrays correctamente.

Arrays: Acceso con índices y punteros

- Los dos bucles siguientes ponen a 0 los elementos de un array:

Con índices:

```
int arr[10], i;

for(i=0; i<10; i++)
    arr[i]=0;
```

Se realizan multiplicaciones cuando se ejecuta.

Con punteros:

```
int arr[10], * ip;

for(ip=arr; ip<arr+10; ip++)
    *ip=0;
```

Se realizan sumas cuando se ejecuta.

- **El acceso con punteros es frecuentemente más rápido que con índices**.
- La mejora en rapidez es máxima cuando se accede al array de forma secuencial (a todos los elementos uno tras otro).
- Si el acceso es aleatorio (a cualquier elemento) con punteros no es más rápido que con índices.
- **<u>DESVENTAJA</u>: La notación se puede complicar usando punteros**, y la claridad del programa puede empeorar.

Memoria Dinámica: `malloc()` y `free()`

- A veces no se sabe **cuánta memoria** se necesita para un array.
  - **SOLUCIÓN: Crear un array con memoria dinámica.**
  - Se declara un puntero y se reserva la memoria necesaria cuando el programa se está ejecutando.
- **FUNCIÓN para Asignar Memoria Dinámicamente: `malloc()`**
  - ***Argumento***: Número de bytes que se quieren reservar (`unsigned int`).
  - ***Devuelve***: Puntero a la zona de memoria reservada (de tipo `void *`).
    - Si no hay suficiente memoria devuelve **NULL**.
    - **SIEMPRE** hay que comprobar si la función devuelve o no **NULL**.
  - ***Útil***: Usar `sizeof()` y usar conversión de tipos para el puntero.
  - **Ejemplo:** Declaramos un puntero a entero y le asignamos la dirección de memoria de un bloque de 10 bytes. El valor devuelto por `malloc()` se asigna al puntero tras efectuar una conversión de tipo adecuada:

```
int *a;
a = (int *) malloc (10)
```

- **FUNCIÓN para Liberar Memoria: `free()`**
  - ***Argumento***: Puntero a la zona de memoria que se quiere liberar.
- Las funciones `malloc()` y `free()` están en `stdlib.h`

Memoria Dinámica: Ejemplo

- **EJEMPLO**: Asignación de memoria dinámica a un entero.
  - **SIEMPRE** se debe comprobar que se le ha asignado al puntero la memoria requerida. En otro caso se puede estar escribiendo en memoria no asignada.

SALIDA EN PANTALLA:

Dirección de a: FFF4. Valor de a: 5

Dirección b: 05E2. Valor al que apunta: 6

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

void main() {
  int a, *b;
  b = (int *) malloc(sizeof(int)); /* Reserva Mm para UN entero */
  if (b == NULL)
     printf("No hay memoria suficiente.");
  else {
     a=5;
     *b=6;
     printf("Dirección de a: %p. Valor de a: %d\n",
       &a, a);
     printf("Dirección b: %p. valor al que apunta: %d\n",
       b, *b);
     free(b); /* Importante: Liberar la memoria */
  }
}
```

Arrays Dinámicos: Ejemplo

- **EJEMPLO**: Programa que lee y escribe los valores de un array de reales **V**. El número de valores del array se conoce durante la ejecución (es un array dinámico). Se reserva toda la memoria con **malloc(***tama***)**.

```
#include <stdio.h>
#include <stdlib.h>
void main() {
  float *V, *ptr; /* ptr es para acceder al array V por punteros */
  int N, i;
  printf("Número de valores: "); scanf("%d",&N);
  if ((V=(float *) malloc(N*sizeof(float))) == NULL) {
    printf("No hay suficiente memoria.\n");
  else {
    for(i=0,ptr=V; i<N; i++,ptr++) {
       printf("Valor >> ");
       scanf("%f",ptr /* o &V[i] */);
    }
    for(i=0,ptr=V;i<N;i++,ptr++)
       printf("\nValor %d >> %f", i, *ptr /* o V[i] */);
    free(V);
  }
}
```


Arrays Dinámicos: realloc() y Ejemplo

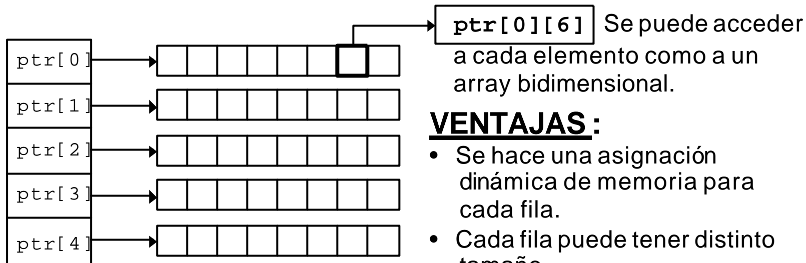
- **realloc(***punt,nuevotama***)**: Cambia el tamaño del bloque apuntado por *punt*. El nuevo tamaño puede ser mayor o menor y no se pierde la información que hubiera almacenada (si cambia de ubicación se copia).
- **EJEMPLO**: Programa que lee y escribe los valores de un array **V** de reales. El número de valores se conoce durante la ejecución. La memoria no se reserva toda a la vez sino elemento a elemento.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>  /* para getch() */
void main() {
  float *V=NULL; int N=0,i; char c;
  do {
     V=(float *)realloc((float *)V,(N+1)*sizeof(float));
     printf("\nDame valor >>"); scanf("%f",&V[N]);
     printf("Quieres introducir otro valor? (S/N) >> ");
     c=getch();
     N++
  } while (c=='s' || c=='S');
  for(i=0;i<N;i++) printf("\nValor %d >> %f\n",i,V[i]);
  free(V)
}
```

Arrays de Punteros

- **EJEMPLO:** Array de 5 punteros, cada uno de los cuales apunta a un array de 8 reales:
  - Es como un **array bidimensional**, con dos diferencias:
    - Cada fila puede tener distinto número de elementos.
    - Las filas no tienen que estar de forma consecutiva en memoria.

```
float *ptr[5]; /* Array de 5 punteros a float */
for(i=0;i<5;i++)
   ptr[i] = ( float * ) malloc(8 *  sizeof(float));
```



Se puede acceder a cada elemento como a un array bidimensional.

VENTAJAS :

- Se hace una asignación dinámica de memoria para cada fila.
- Cada fila puede tener distinto tamaño.

Tipos Enumerados

- En **C**, un **Tipo Enumerado** consiste en una sucesión de **constantes enteras con nombre** que especifica todos los valores válidos para cierta variable.
  - Simplifican la programación haciendo que los programas sean más legibles, ya que usarían los símbolos (los nombres válidos).
  - **Sintaxis:** `enum nombre_tipo {lista_nombres} variables;`
  - **Ejemplo:** `enum semana {L, M, X, J, V, S, D} dia;`
    - Con ese ejemplo es correcto usar: `dia = S;` `if (dia == X) ...`
  - Cada **símbolo** de la enumeración tiene un **valor entero:** Se empieza en cero y para cada símbolo se suma 1: `L` vale `0`, `M` vale `1`, `X` vale `2`...
  - Esos valores pueden alterarse: `enum semana {L, M, X, J, V, S=10, D} dia;`
    - Así, `S` vale `10` y `D` vale `11`: El orden de los valores se mantiene.
  - Los símbolos **NO pueden escribirse** directamente como si fueran cadenas de caracteres: **Son constantes enteras**.
  - Usualmente el sistema **NO genera error** si no se respetan las restricciones de un tipo enumerado: Es responsabilidad del programador usarlo correctamente.

Clases de Almacenamiento de Variables

- **Clases de almacenamiento en C:** En una declaración la clase de almacenamiento puede especificarse antes de la declaración:
  - `auto`, **Variables Automáticas:** Son las **variables normales**, que se crean cuando se declaran y se destruyen cuando se termina la función en la que están declaradas (o el programa si son globales).
    - Es equivalente declarar `int i;` que `auto int i;`
  - `extern`, **Variables Externas:** Se refiere a que se trata de variables declaradas en un **módulo(fichero) distinto** a donde aparece la declaración con `extern`. •**Ej.:** `extern int i; /* Variable ya declarada en otro módulo */`
  - `register`, **Variables Registro:** Son las variables que intentan ser almacenadas en algún **registro del procesador**, para que las operaciones sobre ellas sean más rápidas. Se usará en variables sobre las que recaigan múltiples operaciones (control de bucles...). No puede abusarse de este tipo de variables ya que, en tal caso, algunas variables no se almacenarán en registros. •**Ej.:** `register int i;`
  - `static`, **Variables Estáticas:** Variables que se crean la primera vez que se ejecuta su declaración y no se destruyen al finalizar su función. Son como variables globales, aunque sólo son vistas en la función en la que se declaran.
    - La segunda vez (y las siguientes) que se llame a una función con una variable estática, la función "recordará" el valor que obtuvo la variable en la ejecución anterior, ya que la variable NO se destruyó al finalizar dicha ejecución anterior. •**Ej.:** `static int i=9; /* Valor inicial */`
    - Una variable estática siempre debe **inicializarse** al declararse.
    - Si se usa `static` en una **var. global** será vista sólo en ese fichero.

Operadores a Nivel de Bits

- Los **operadores a Nivel de Bits** operan sobre la representación en binario de sus operandos.
  - Sólo pueden usarse sobre los tipos `int` y `char` y sus modificaciones (no pueden usarse sobre tipos reales, `void`...).
- **Operadores con sus significados, un ejemplo (con `int a=5, b=3`) y su resultado:**

```
a º 5 º 0101
b º 3 º 0011
```

Opdor.	Significado	Ejemplo	Resultado
&	**Y lógico**	`a & b`	1
|	**O lógico**	`a	b`	7
^	**XOR lóg.**	`a ^ b`	6
~	**Complemento a 1**	`~b`	-4
>>	**Desplazamiento a la derecha**	`a >> 1`	2
<<	**Desplazamiento a la izquierda**	`b << 2`	12

  - El desplazamiento a la izquierda (resp. a la derecha) equivale a la multiplicación (resp. división entera) por 2 tantas veces como indique el número de bits desplazados (segundo operando).
  - Con estos operadores es simple acceder a un bit concreto de un dato.

Campos de Bits

- **Campos de Bits:** Son un método para acceder fácilmente a un bit individual dentro de un byte, o a un grupo de varios bits.
  - Puede usarse para almacenar distintos valores en un mismo byte.
    - Ejemplo: Varias variables lógicas (1 Verdad, 0 Falso).
  - En las transmisiones es típico transmitir información codificada en los bits de un byte.
    - Ejemplo: Para informar que la impresora no tiene papel, o que cierto dispositivo no está disponible...
  - Los campos de bits pueden usarse junto con campos normales.
- **Declaración:** Es similar a las estructuras, donde el tipo del campo sólo puede ser `int`, `unsigned` o `signed`. Además, se le añade el número de bits de cada campo:

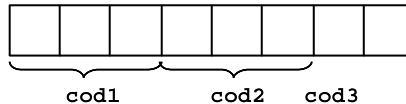
```
struct nombre_estruct {
  tipo_1 nombre_1 : longitud_1;
  tipo_2 nombre_2 : longitud_2;
  . . .
}
```

  - Si la longitud (en bits) es 1, entonces el tipo sólo puede ser `unsigned`.

Campos de Bits

- **Ejemplo:** Con la variable `V` se puede acceder a los 3 primeros bits, a los 3 segundos bits y a al séptimo bit individualmente.

```
struct nombre_estruct {
  int      cod1 : 3;
  unsigned cod2 : 3;
  unsigned cod3 : 1;
} V;
```



- **Observaciones:**
  - La variable `V` sólo ocupa 1 byte (8 bits) y el octavo bit no es accesible (porque no es necesario, por ejemplo).
    - Sin usar un campo de bits esto hubiera ocupado 3 bytes como mínimo. Imagine el ahorro en un array de estructuras de este tipo.
  - Campos de la variable `V`:
    - `V.cod1` es un entero de 3 bits, por lo que su rango es [–4,3] (utiliza una representación en complemento a 2).
    - `V.cod2` es un entero positivo de 3 bits. Su rango es [0,7].
    - `V.cod3` es un entero positivo de 1 bits. Su rango es [0,1].
      - Este último campo es usual para variables lógicas (booleanas) que sólo admiten dos valores: Verdad (1) o Falso (0).

Uniones

- **Uniones:** Es una forma de hacer que diferentes variables compartan la misma zona de la memoria.
  - Su declaración tiene exactamente la misma sintaxis que las estructuras, excepto que se usa la palabra **`union`** en vez de **`struct`**.
  - **Ejemplo:**

```
union tipo_union{
  int i;
  char ch;
} V;
```

i	
ch	

    - Si a **`V.i`** se le asigna un valor, el primer byte de ese entero puede ser visto desde **`V.ch`**.
  - **Ejemplo:** Para acceder de forma individual al segundo byte de **`V.i`** podríamosdeclarar **`ch`** como un array de dos posiciones:

```
union tipo_union{
  int i;
  char ch[2];
} V;
V.i=258;  /* En binario: 00000001 00000010 */
printf("%i-%i", V.ch[1], V.ch[0]);
```

i	
ch[0]	ch[1]

    - Escribe: **`1-2`**. Observe que en un PC los bits se almacenan desde el menos significativo al más significativo, (contrario a la intuición) y, por eso, mostramos primero el segundo byte.

Uniones

- **Características de las Uniones:**
  - Ocupan tanto espacio en **Memoria** como el **mayor** de sus campos.
    - Las estructuras ocupan tanta memoria como la suma de todos sus campos.
  - Una vez definida una **unión**, se pueden **declarar variables** de forma similar a las estructuras.
    - **Ejemplo:** **`union tipo_union tu1, tu2;`**
  - Para **acceder a los campos** se accede, igual que en las estructuras, usando el operador punto (si tenemos una unión) y el operador flecha (si tenemos el puntero a una unión).
  - Igual que ocurre con los campos de bits, el código puede ser **dependiente de la máquina:**
    - Existen máquinas que representan los datos de izquierda a derecha y de derecha a izquierda.
  - Una unión puede tener **más de dos campos:** Todos los campos comparten la memoria.
    - Esos campos pueden ser de cualquier tipo, incluyendo arrays, estructuras, campos de bits...

Uniones y Campos de Bits: Ejemplo

- **Ejemplo:** Acceder a los 8 bits de un carácter uno a uno, para escribir la representación interna de ese carácter.
  - El programa muestra los caracteres desde la `'A'` a la `'E'`.

```
union {
    char ch;
    struct { unsigned a:1;
             unsigned b:1;
             unsigned c:1;
             unsigned d:1;
             unsigned e:1;
             unsigned f:1;
             unsigned g:1;
             unsigned h:1; } s;
  } var_char;

for(var_char.ch='A'; var_char.ch<='E'; var_char.ch++)
  printf("\n%c - %i - %u%u%u%u%u%u%u%u.",
    var_char.ch, var_char.ch,
    var_char.s.h, var_char.s.g, var_char.s.f, var_char.s.e,
    var_char.s.d, var_char.s.c, var_char.s.b, var_char.s.a);
}
```

Salida por Pantalla:

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


Uniones y Campos de Bits: Ejemplo

- **Ejemplo:** Separar un número entero en partes de 4 bits cada una.
  - Observe que el resultado se imprime en orden inverso para obtener el número en el orden usual (la parte más significativa a la izquierda).
  - Observe que esto es similar a pasar un número positivo a hexadecimal (agrupamos los bits de 4 en 4), excepto que no escribe en letra los valores a partir de 10.
    - Con los números negativos tenga en cuenta que se usa la representación en complemento a 2.

```
union separa{
  int numero;
  struct de4en4 {
    unsigned a1:4;
    unsigned a2:4;
    unsigned a3:4;
    unsigned a4:4;
  } cuatro;
};
union separa dual;

dual.numero = 65;
printf("\n%u-%u-%u-%u.",dual.cuatro.a4,dual.cuatro.a3,
                        dual.cuatro.a2,dual.cuatro.a1);
```

Salida por Pantalla:

```
0-0-4-1.
```

Tenga en cuenta que el número **65** en binario se escribe como:

`0000 0000 0100 0001`

Interrupciones Software

- Una **Interrupción** es un tipo especial de instrucción que ***interrumpe*** la ejecución de un programa:
  - Una **Interrupción** hace que se pare la ejecución del programa actual, se salte a la ejecución de una *rutina* de tratamiento de la interrupción y, finalmente, vuelve para continuar el programa que había sido interrumpido.
  - Las **interrupciones software**, que suelen numerarse con números en hexadecimal, pueden ser ejecutadas desde un programa y sirven para solicitar que el programa efectúe determinadas tareas.
    - En un **PC** esas tareas pueden ser: imprimir la pantalla (interrupción 5h), operaciones de E/S de vídeo (10h), de E/S de disco (13h), de E/S del puerto serie (14h), de E/S del teclado (16h), de E/S de impresora (17h), operaciones de hora y fecha (1Ah), de control del ratón (33h)...
- En un **PC**, la función típica para ejecutar una **interrupción software** está en **`dos.h`**:

```
int int86 (int Num_Interrupcion,
           union REGS *Entrada,
           union REGS *Salida);
```

  - **`Num_Interrupcion`** es la **interrupción** que deseamos ejecutar.
  - **`Entrada`** son los datos de **entrada** a dicha interrupción.
  - **`Salida`** son los datos de **salida** de dicha interrupción.
  - La función **devuelve** el valor del registro **AX** tras la interrupción.

Interrupciones Software

- El tipo **`union REGS`** está definido en **`dos.h`** de forma que pueda accederse a los registros del procesador de forma completa (como **`ax`**) o byte a byte individualmente (**`al`** es la parte baja y **`ah`** es la parte alta, ***l**ow*/***h**ight*):

```
struct WORDREGS {
    unsigned int ax, bx, cx, dx, si, di, cflag, flags;
};
struct BYTEREGS {
    unsigned char al, ah, bl, bh, cl, ch, dl, dh;
};
union REGS {
    struct  WORDREGS  x;
    struct  BYTEREGS  h;
};
```

- **<u>Ejemplo</u>:** Borrar parte de la pantalla (en este ejemplo borra la pantalla completa):

```
union REGS r;

r.h.ah=6; r.h.al=0;
r.h.ch=0;  /* Fila inicial */
r.h.cl=0;  /* Columna inicial */
r.h.dh=24; /* Fila final */
r.h.dl=79; /* Columna final */
r.h.bh=7;  /* Color de borrado */
int86(0x10, &r, &r);
```

Suele utilizarse la misma variable de **entrada** también para la **salida**

E/S por Ficheros (*Files*): Operaciones Básicas

- Para operar con un **Fichero o Archivo** hay que seguir **3** pasos:
  - 1. **Abrir el Fichero:** Prepara un fichero para poder utilizarlo.
    - La **función** `fopen()` (de `stdio.h`) devuelve un puntero a un fichero (`FILE *`), que es una estructura con la información que necesita el sistema para acceder al fichero (nombre, estado, posición actual...).
      - `fopen()` devuelve `NULL` si no se puede abrir el fichero (no existe, disco lleno, disco protegido...).
    - El **primer argumento de** `fopen()` es una cadena de caracteres con el **nombre** (y la ruta) del fichero que se desea abrir.
    - El **segundo argumento de** `fopen()` es una cadena de caracteres con el **modo** con el que se desea abrir un fichero:

`r`	Abre un fichero ya existente sólo para lectura (*read*).
`w`	Crea un fichero para escritura (*write*). Si ya existe, lo borra.
`a`	Abre un fichero para añadir (escribir) datos al final (*append*). Si no existe, lo crea.
`r+`	Abre un fichero existente para actualización (lectura y escritura).
`w+`	Crea un fichero para actualización (lectura y escritura). Si ya existe, lo borra.
`a+`	Abre un fichero para actualización (lectura y escritura) al final. Si no existe, lo crea.

- Puede añadirse una `'t'` para indicar que el fichero es de **texto** y una `'b'` para indicar que el fichero es **binario**.
  - Los **ficheros de texto** ocupan más espacio que los **binarios**, pero pueden visualizarse (y modificarse) con cualquier editor de texto.

E/S por Ficheros (*Files*): Operaciones Básicas

- 2. **Operar con el Fichero:**
  - Las **operaciones típicas** con un fichero son **Leer y Escribir** y para estas operaciones hay muchas funciones que veremos a continuación.
  - Existen **otras operaciones** que pueden resultar útiles y que también veremos a continuación. Estas operaciones son, por ejemplo, situarnos en una posición concreta del fichero (para leer o escribir allí), ver si hemos llegado al final del fichero, situarnos al principio del fichero...
- 3. **Cerrar el Fichero:** Finaliza la utilización de un fichero. Si no se hace se pueden perder los datos del fichero.
  - La función `fclose()` tiene como argumento un puntero a fichero (`FILE *`) y cierra un fichero previamente abierto con `fopen()`.
- **Ej.:**

```
FILE * fich; /* fich es un puntero a fichero */
fich = fopen("datos.txt", "rt");
if (fich==NULL)
   puts("\n- No se puede abrir el fichero.");
else { /* Operaciones de E/S en el fichero */
       ...
       fclose(fich);
}
```

E/S por Ficheros (Files): Funciones de E/S de `stdio.h`

- `int putc(int c, FILE *fich);`
  - Escribe el carácter `c` en el fichero `fich`.
  - Devuelve el carácter escrito, o bien, `EOF` si hubo algún error.
    - `EOF` (*End Of File*, Fin De Fichero) es una constante simbólica definida también en `stdio.h`.
- `int getc(FILE *fich);`
  - Lee un carácter del fichero `fich`.
  - Devuelve el carácter leído, o bien, `EOF` si hubo algún error (fin de fichero...).
- `int fprintf(FILE * fich, const char *format [, args]);`
  - Similar a la función `printf()` pero el resultado lo escribe en el fichero `fich`.
  - Ejemplo: Para escribir un texto con una variable real en el fichero salida: `fprintf(salida,"Jamones %f", peso);`
- `int fscanf(FILE * fich, const char *format [, direcs]);`
  - Similar a `scanf()` pero el origen de la lectura es el fichero `fich`.

E/S por Ficheros (Files): Funciones de E/S de `stdio.h`

- `int fputs(const char *cadena, FILE *fich);`
  - Escribe la cadena de caracteres en el fichero, sin añadir el carácter de nueva línea.
  - Devuelve el último carácter escrito o `EOF` si hubo error.
- `char *fgets(char *s, int n, FILE *fich);`
  - Lee caracteres del fichero y los almacena en `s`. Para de leer cuando lee `n-1` caracteres, el carácter de nueva línea o el carácter `EOF`.
  - Devuelve la cadena leída o `NULL` si hubo error.
- `size_t fwrite(void *p, size_t size,size_t n, FILE *f);`
  - Escribe, en un fichero BINARIO, `n` elementos de tamaño `size` (en bytes). Se supone que esos elementos están en la memoria a la que `p` apunta.
  - Permite la escritura de un bloque de datos apuntado por `p`.
  - El tipo `size_t` es `unsigned`.
  - Se escribirán un total de (`size*n`) bytes en BINARIO.
  - El puntero `p` puede ser la dirección de una variable, o bien, un bloque de memoria asignado dinámicamente (con `malloc()`, por ejemplo).
  - Ejemplo: Para escribir un array `V` de 8 enteros en el fichero `f`, usar:

    ```
    fwrite(V, sizeof(int ), 8, f);
    ```
- `size_t fread(void *p, size_t size,size_t n, FILE *f);`
  - Lee, de un fichero BINARIO, `n` elementos de tamaño `size` (en bytes). Los almacena en la memoria a la que `p` apunta (suponiendo que caben).

E/S por Ficheros (Files):
Funciones Útiles de `stdio.h`

- `int feof(FILE *fich);`
  - Devuelve un valor distinto de cero si se alcanzó ya el final del fichero (EOF, End Of File) en la última lectura.
  - Devuelve cero si no se llegó al final.
- `int ferror(FILE *fich);`
  - Devuelve un valor distinto de cero si se produjo algún error en la última operación con el fichero.
  - Se debe llamar a esta función después de cada operación para garantizar que el proceso se realiza correctamente.
- `void rewind(FILE *fich);`
  - Inicializa el indicador de posición del archivo al principio del mismo (rebobinar).
- `int fseek(FILE *fich, long desplazamiento, int donde);`
  - Desplaza el indicador de posición del fichero tantos bytes como indique `desplazamiento`, a partir de la posición indicada por el argumento `donde`, que puede tomar 3 valores: `SEEK_SET` (a partir del inicio), `SEEK_CUR` (a partir de la posición actual) y `SEEK_END` (a partir del final del fichero).
  - Devuelve 0 si el desplazamiento se efectuó con éxito.
  - En ficheros de texto puede producir errores.

E/S por Ficheros (Files):
Funciones Útiles de `stdio.h`

- `long ftell(FILE *fich);`
  - Devuelve el valor actual del indicador de posición de un archivo binario. Este valor puede usarse en la función `fseek()` para volver a la posición anterior.
  - Devuelve `-1L` (`-1` como `long`) si se produce un error.
  - Es similar a la función `fgetpos()`, la cual tiene una función inversa que es `fsetpos()`.
- `int fflush(FILE *fich);`
  - Si el fichero es un flujo (*stream*) abierto para escritura, entonces `fflush()` escribe en el fichero el contenido del *buffer*.
  - Si el fichero está abierto para lectura, entonces `fflush()` borra el contenido del *buffer*.
  - Devuelve cero si la operación se realizó correctamente y `EOF` si se produjo algún error.
  - Si se aplica sobre `stdin` borra el *buffer* de teclado.
- `int flushall(void);`
  - Limpia los *buffers* de todos los ficheros abiertos, devolviendo el número de ficheros abiertos (incluyendo los abiertos automáticamente).
- `int fcloseall(void);`
  - Cierra todos los ficheros excepto los abiertos automáticamente.

`stdin` *y* `stdout`

- **Son dos ficheros que se Abren automáticamente cuando un programa empieza a ejecutarse y se Cierran automáticamente al finalizar el programa:**
  - **`stdin` (*standard input*): Es la Entrada estándar.**
  - **`stdout` (*standard output*): Es la Salida estándar.**
- **Normalmente se refieren a la consola (teclado y pantalla), pero desde el S.O. (Sistema Operativo) pueden redirigirse hacia otros dispositivos: ficheros, impresora...**
  - **Esto puede hacerse utilizando las redirecciones > y <, además de la tubería |, en una orden del S.O.**
- **Son punteros a archivos y, por tanto, pueden utilizarse con las funciones de manejo de ficheros:**
  - **`stdin` en funciones de Lectura.**
  - **`stdout` en funciones de Escritura.**
- **Existen otros ficheros de apertura y cierre automáticos:**
  - **`stderr` es el dispositivo estándar para la escritura de errores (usualmente es también la pantalla).**
  - **`stdprn` es el dispositivo estándar de impresora.**
  - **`stdaux` es el dispositivo auxiliar estándar.**

Tipos Abstractos de Datos: Introducción

- **Tipo Abstracto de Datos, TAD o TDA** (*Abstract Data Type*, ADT)**:**
  - Consiste en un **Conjunto de Valores** y unas **Operaciones** que pueden hacerse sobre esos valores.
  - El término **TAD** se refiere al **concepto matemático** básico que define el tipo de datos: No interesa la eficiencia del espacio o del tiempo, ni cómo se almacenan los datos, ni cómo se implementarán las operaciones.
  - **Ejemplos:** Existen **TAD** cuyas estruturas de datos y operaciones ya están definidas por el **Lenguaje C** estándar:
    - **Arrays:** Es una sucesión de elementos indexados y tiene unas operaciones bien definidas que se pueden hacer sobre sus elementos.
    - **Ficheros:** Es un tipo de dato específico (**`FILE *`**) y tiene también sus operaciones claramente definidas (abrir, cerrar, leer, escribir...)
    - **Enteros:** Estrictamente hablando, los números enteros también son TAD. Un entero tiene un valor y unas operaciones (suma, resta, multiplicación, división...).
  - Un programador puede **crear un TAD:** Crear una estructura de datos que almacene los **valores** e implementar **operaciones** sobre ellas.
    - **Ejemplo:** Para utilizar **números complejos** podemos definir que un número complejo es una estructura con dos valores y posteriormente definir operaciones sobre ese tipo de dato: creación de un número complejo, suma de complejos, multiplicación, división...

Tipos Abstractos de Datos: *Introducción*

- **Compilación Separada:** En general, es recomendable que cuando se crea un **TAD**, se implemente en un **módulo separado**, aprovechando así las ventajas de la **compilación separada** (reducir tiempo de compilación, poder re-utilizar fácilmente ese **TAD** en otros programas...). Para ello, crear **dos ficheros:**
  - **Fichero `.h` de cabeceras:** En este fichero se definen las estructuras de datos del **TAD** y se establecen los prototipos o cabeceras (*headers*) de las funciones que implementan las operaciones.
  - **Fichero `.c` con las operaciones:** Este fichero, que incluirá al anterior con `#include`, implementará las operaciones.
- **Independencia de Datos:** Una carácterística importante de un programa que utilice un **TAD** creado por el programador es que si se cambia cualquier aspecto de la **estructura de datos** que alberga el **TAD** o de la **implementación** de sus operaciones, el programa debería seguir funcionando correctamente.
  - Las **operaciones sobre el TAD** deben tener una interfaz conocida (nombre y tipo de la función, número y tipo de sus argumentos...).
  - O sea, las **funciones que necesiten el TAD** deben usarlo ***exclusivamente*** a través de las operaciones definidas para el **TAD**.
  - Es importante **definir correcta y completamente un TAD** antes de implementarlo: Valores, operaciones necesarias, posibles errores en las operaciones y cómo se comunicarán los mismos...
  - Un **TAD puede tener restricciones. Ejemplos:** Un array tiene un número de elementos máximo, no podemos almacenar números enteros infinitamente grandes... Se deben controlar todas ellas y gestionar los errores que puedan producirse.

Tipos Abstractos de Datos: *Introducción*

- **Hay infinitos TAD posibles**, pero hay algunos que son muy típicos y suelen utilizarse mucho en programación para solucionar multitud de problemas.
  - Recuerde que cada **TAD** pueden implementarse de varias formas distintas (utilizando o no memoria dinámica...).
  - **Ejemplos:**
    - **Lista:** Es una sucesión de elementos en cierto orden. El número de elementos puede variar.
    - **Pila (estructura LIFO,** *Last In First Out***):** Es un conjunto de elementos del que sólo podemos leer el último e insertar en la última posición.
    - **Cola (estructura FIFO,** *First In First Out***):** Es un conjunto de elementos del que sólo podemos leer el primero e insertar en la última posición.
    - **Árbol:** Estructura con distintos nodos o elementos, tal que cada elemento puede tener a otros elementos como “hijos”. Nodos “hoja” son los que no tienen hijos y nodo “raíz” es aquel que no tiene “padre” (es el nodo principal). Se llaman árboles binarios si se admite un máximo de dos hijos por nodo.
    - **Grafo:** Es un conjunto de elementos que pueden conectarse entre sí de cualquier manera (como las carreteras que conectan ciertas ciudades).

Estructuras de Datos Dinámicas

- **Son aquellas cuyo tamaño (longitud, número de elementos...) varía en tiempo de ejecución.**
  - Las más famosas son: Listas simplemente enlazadas (con y sin cabecera), Listas doblemente enlazadas (con y sin cabecera), Pilas, Colas, Árboles y Grafos.
  - Usan estructuras autorreferenciadas para definir un nodo o elemento.
  - Así, cada elemento puede referenciar a otro u otros usando la dirección de memoria de cada uno.
  - **Ejemplo: Nodo de un árbol binario:**
    - Un árbol es un puntero a un nodo.
    - Función para crear un nodo del árbol:

```
struct nodo {
  char etiqueta[20];
  struct nodo *hijo_izda;
  struct nodo *hijo_dcha;
};

typedef struct nodo *arbol;
```

```
arbol CreaNodoVacio (){
  arbol arb;
  if ( (arb=(arbol) malloc( sizeof( struct nodo))) == NULL)
     return NULL;
  arb->etiqueta[0]='\0';  /* Cadena vacía            */
  arb->hijo_izda  = NULL; /* Sin hijo a la izquierda */
  arb->hijo_dcha  = NULL; /* Sin hijo a la derecha   */
  return arb;
}
```


EDD: Lista Dinámica de Números

- **Lista simplemente enlazada sin cabecera:** Sin cabecera indica que no existe un elemento especial en la primera posición, i.e., que el primer nodo es el primer elemento de la lista.
  - **Declaración de TIPOS de DATOS:**

```
/* Declaración de tipos */
struct element {
  long num;
  struct element *sig;
};

typedef struct element *lista;
```

Por supuesto, en vez de un único número cada elemento puede incluir cualquier cantidad de datos.

  - **Funciones primitivas importantes:**

```
/* Inicializa la lista */
void inicializar_lista(lista *list){
  *list=NULL;
}

/* Devuelve TRUE si la lista está vacía. */
int lista_vacia(lista list){
  if (list) return 0;
  return 1;
}
```

EDD: Lista Dinámica de Números

```
/* Devuelve la longitud de la lista (núm. de elems).  */
long long_lista(lista list){
  long i=0;

  while (list) {
    list=list->sig;
    i++;
  }
  return i;
}
/*******************************************************/
/* Devuelve en e, el elemento de la posición pos de list. */
/* Si no existe esa posición,devuelve -1. En otro caso 0.*/
int leer_element(lista list, long pos, struct element *e){
  long i=1;
  if ( lista_vacia(list) || pos<1)
     return -1;

  while (list && i<pos) {
    list=list->sig;
    i++;
  }
  if (list){
     *e=*list;
     return 0;
  }
  return -1;
}
```


EDD: Lista Dinámica de Números

```
/* Imprime un elemento e por la Salida estándar.        */
void Imprimir_elemento (struct element e){
  printf("%li",e.num);
}
/*******************************************************/
/* Muestra todos los elementos de list por su orden.     */
void mostrar_lista(lista list){
  struct element e;
  unsigned i=1,tama=long_lista(list);
  while (i<=tama) {
    printf("\nElemento %u: ",i);
    leer_element(list,i++,&e);
    Imprimir_elemento(e);
  }
}
/*******************************************************/
/* Actualiza la posición pos de la lista list con e      */
/* Devuelve -1 si no existe esa posición.                */
int actualiza_lista(lista list, struct element e, long pos){
  long i=1;
  if (pos<1) return -1;   /* Posición no válida   */
  while (list && i<pos) {  /* Ir a la posición pos */
    list=list->sig;
    i++;
  }
  if (!list) return -1; /* Posición no existe */
  list->num=e.num;      /*Actualización */
  return 0;
}
```

EDD: Lista Dinámica de Números

```
/* Inserta el elemento e en la posición pos de list.   */
/* Si pos<=1 inserta en la primera posición.           */
/* Si pos>longitud_lista, inserta en la última posción.*/
/* Devuelve -1 si no hay memoria suficiente.           */
int insertar_pos ( lista *list, struct element e, long pos){
  lista p, anterior, L=*list;
  long i=1;
  if ((p=(struct element *) malloc(sizeof(struct element)))
       == NULL)
     return -1;
  *p=e;
  if (pos<=1 || lista_vacia(*list)){
    /* Hay que insertar en la posición 1 */
    *list=p;
    p->sig=L;
    return 0;
  }

  while (L && i<pos){ /* Buscar la posición pos */
    anterior=L;
    L=L->sig;
    i++;
  }
  /* Insertar elemento apuntado por p, entre anterior y L */
  anterior->sig=p;
  p->sig=L;
  return 0;
}
```


EDD: Lista Dinámica de Números

```
/* Borra el elemento en la posición pos de la lista list.  */
/* Si la pos=1 borra el primer elemento.                    */
/* Devuelve -1 si no existe la posición:                    */
/*                        pila vacía o pos>long             */
int borrar_elemento(lista *list,long pos){
  lista aux, anterior, L=* list;
  long i=1;
  if ( lista_vacia(*list) || pos<1)
     return -1;
  if (pos==1) { /* Tratamiento para borrar elemento 1 */
     (*list) = (*list)->sig;
     free(L);
     return 0;
  }
  while (L && i<pos){/* Buscar la posición pos */
    anterior=L;
    L=L->sig;
    i++;
  }
  if (L){ /* OJO: Aqui no vale decir: if (i==pos)...*/
     /* Borrar elemento apuntado por L,
        teniendo un puntero al anterior */
     anterior->sig=L->sig;
     free(L);
     return 0;
  }
  return -1; /* La lista tiene menos de pos elementos */
}
```

EDD: Lista Dinámica de Números

```
/* Libera la memoria ocupada por toda la lista.          */
void liberar_lista(lista*list){
  while (*list) {
    borrar_elemento(list,1);
  }
}
/*********************************************************/
/* Devuelve posición de la primera ocurrencia del elemento*/
/* e en la lista list, a partir de posiciónpos(inclusive)*/
/* Esta ocurrencia será considerada por el campo num.    */
/* Devuelve 0 si no ha sido encontrada.                  */
/* Cambiando pos, podremos encontrar TODAS               */
/* las ocurrencias de un elemento en la lista.           */
long posic_lista(lista list, struct element e, long pos){
  long i=1;
  while (list && i<pos) {  /* Ir a la posición pos */
    list=list->sig;
    i++;
  }
  if (!list) return 0;
  while (list && e.num!=list->num) {
    /* Intentar encontrar el elemento */
    list=list->sig;
    i++;
  }
  if (!list) return 0; /* No existe */
  return i;            /* Existe en la posición i */
}
```


Bibliografía

- J.M. Rodríguez, J. Galindo. "**Aprendiendo C**", 3ª Edición. Universidad de Cádiz, 2006.
  - Un libro básico y barato pero muy completo, con multitud de ejemplos propuestos y resueltos en esta tercera edición.
- Kenneth A. Reek. "**Pointers on C**". Addison-Wesley, 1998.
  - Un libro muy bueno, con ejercicios muy interesantes para aprender.
- W. Buchanan ."**C for Electronic Engineering**". Prentice Hall, 1995.
  - Lo mejor es que utiliza ejercicios propios de electrónica.
- Herbert Schildt "**C Manual de referencia**". McGraw Hill, 1990.
  - Un libro mítico para tenerlo como referencia.
- P.J. Sánchez, J. Galindo *et al*. "**Ejercicios Resueltos de Programación C**". Servicio de Publicaciones de la Univ. de Cádiz, 1997.
- Peter Norton, "**Introducción a la Computación**". McGraw-Hill, 1995.
- James L. Antonakos, Kenneth C. Mansfield Jr., "**Programación Estructurada en C**". Prentice Hall, 1997.
- Y. Langsam, M.J. Augenstein, A.M. Tenenbaum, "**Estructuras de Datos con C y C++**". Prentice Hall, 1997.
- F. Javier Ceballos, "**Curso de Programación C/C++**". Ra-ma, 1995.