Introducción al Lenguaje de Programación C

Félix García Carballeira

1999

Índice

1. Introducción 5
2. Fundamentos de C 15
3. Operadores y expresiones 30
4. Sentencias de control 51
5. Funciones y programación estructurada 79
6. Punteros y ámbito de las variables 97
7. Cadena de caracteres 123
8. Vectores 136
9. Estructuras de datos 156
10. Entrada/Salida 181
11. Aspectos avanzados de C 192
12. Herramientias para el desarrollo de programas en UNIX 215

Bibliografía

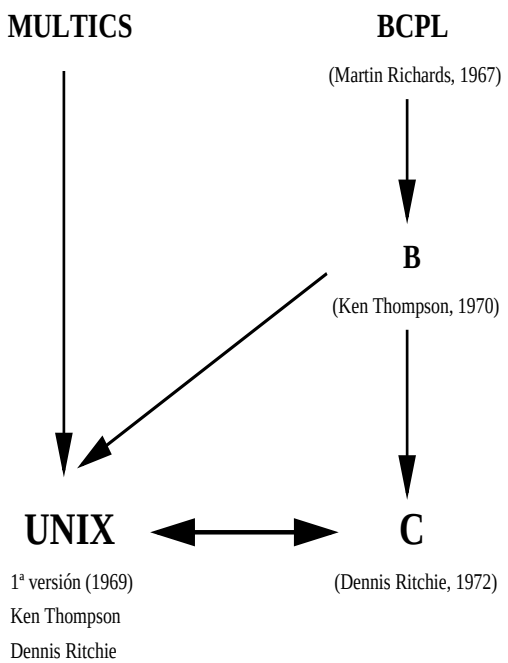
- *Programación Estructurada en C*, J. L. Antonakos, K. C. Mansfiled Jr. Prentice-Hall, 1997.
- *The C programming Language*, B. Kernighan, D. Ritchie. Second Edition. Prentice-Hall, 1988.
- *C a Reference Manual*, S. P. Harbison, G. L. Steele Jr. Foruth Edition, Prentice-Hall, 1992.
- *On to C*, Patrick Henry Winston. Addison-Wesley, 1994.
- *The UNIX Programming Environment*, B. Kernighan, R. Pike. Second Edition, Prentice-Hall, 1988.
- *UNIX Unbounded A Beginning Approach*, Amir Zfzal. Prentice-Hall, 1995.

INTRODUCCIÓN AL LENGUAJE C

Historia de C

- Muchas ideas provienen de BCPL (*Martin Richards*, 1967) y de B (*Ken Thompson*, 1970).
- C fue diseñado originalmente en 1972 para el SO UNIX en el DEC PDP-11 por *Dennis Ritchie* en los Laboratorios Bell.
- Primer Libro de referencia de C: *The C Programming Language* (1978) de Brian Kernighan y Dennis Ritchie.
- En 1989 aparece el estándar ANSI C.
- En 1990 aparece el estándar ISO C (actual estándar de C). WG14 se convierte en el comité oficial del estandar ISO C.
- En 1994 WG14 crea las primeras extensiones a ISO C.
- En 1983 aparece C++ (orientado a objetos).

Historia de C y relación con UNIX



Características de C

- Lenguaje de propósito general ampliamente utilizado.
- Presenta características de **bajo nivel**: C trabaja con la misma clase de objetos que la mayoría de los computadores (caracteres, números y direcciones) $\Longrightarrow$programas eficientes.
- Estrechamente asociado con el sistema operativo UNIX:
  - UNIX y su software fueron escritos en C.
- Es un lenguaje adecuado para *programación de sistemas* por su utilidad en la escritura de sistemas operativos.
- Es adecuado también para cualquier otro tipo de aplicación.
- Lenguaje pequeño: sólo ofrece sentencias de control sencillas y funciones.

Características de C (II)

- No ofrece mecanismos de E/S (entrada/salida). Todos los mecanismos de alto nivel se encuentran fuera del lenguaje y se ofrecen como funciones de biblioteca.
- Programas portables: pueden ejecutarse sin cambios en multitud de computadores.
- Permite programación estructurada y diseño modular.

Inconvenientes de C

- No es un lenguaje *fuertemente tipado*.
- Es bastante permisivo con la conversión de datos.
- Sin una programación metódica puede ser propenso a errores difíciles de encontrar.
- La versatilidad de C permite crear programas difíciles de leer.

```
#define _ -F<00||--F-OO--;
int F=00,OO=00;
main(){F_OO();printf("%1.3f\n",4.*-F/OO/OO);}F_OO()
{
                  _-_-_-
              _-_-_-_-_-_-_-
           _-_-_-_-_-_-_-_-_-
        _-_-_-_-_-_-_-_-_-_-_-
      _-_-_-_-_-_-_-_-_-_-_-_-_
     _-_-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
     _-_-_-_-_-_-_-_-_-_-_-_-_-_
     _-_-_-_-_-_-_-_-_-_-_-_-_-_
      _-_-_-_-_-_-_-_-_-_-_-_-_
        _-_-_-_-_-_-_-_-_-_-_
            _-_-_-_-_-_-_-
                _-_-_-
    }
```


Entorno de desarrollo de C

- Editor de texto.
- Compilador.
- Ficheros de cabecera.
- Ficheros de biblioteca.
- Enlazador.
- Depurador.

Primer programa en C

```
#include <stdio.h>

main()
{
       /* Primer programa en C */

       printf("Primer programa en C. \n");
       exit(0);
}
```

¿Cómo crear el programa?

- Con un editor de texto (en UNIX: *vi*, *pico*, *emacs*, etc.)
- Grabar el programa en un fichero, `ejemplo.c`.

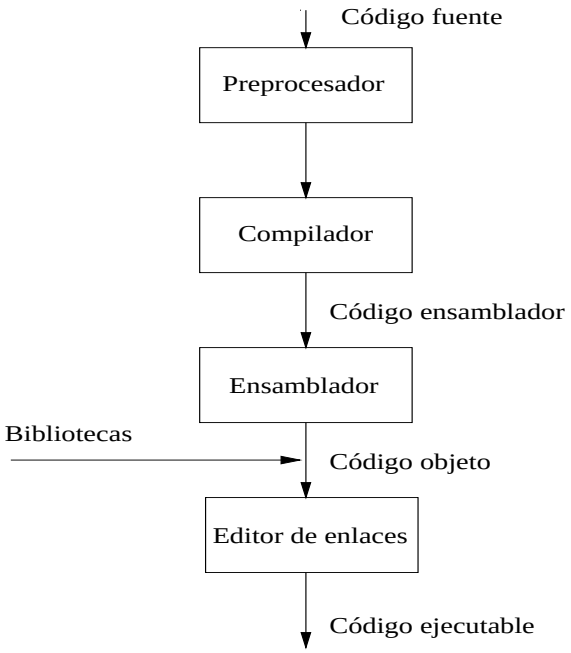
¿Cómo compilar y ejecutar el programa?

- Un **compilador de C** permite:
  - Analizar y detectar errores en el código fuente.
  - Convertir un programa escrito en C en código ejecutable por el computador
- Mandato de compilación básico:

```
cc ejemplo.c
```

  - Genera el código objeto `ejemplo.o`
  - Genera el ejecutable `a.out`
- El programa se ejecuta tecleando `a.out`
- El mandato `cc -c ejemplo.c` genera el fichero objeto `ejemplo.o`.
- El mandato `cc ejemplo.o -o ejemplo` genera el ejecutable `ejemplo`
- En este caso el programa se ejecuta tecleando `ejemplo`.

Modelo de compilación de C



FUNDAMENTOS DE C

Comentarios

- Comienzan con `/*` y finalizan con `*/`
- No se pueden anidar. Dentro de un comentario no puede aparecer el símbolo `/*`.

```
#include <stdio.h>

main()
{
       /*********************************/
       /*     Esto es un comentario     */
       /*********************************/

       printf("Este programa contiene\n");
       printf(ün comentario. \n");
       exit(0);
}
```

Palabras reservadas

auto	do	for	return	typedef
break	double	goto	short	union
case	else	if	sizeof	unsigned
char	enum	int	static	void
const	extern	long	struct	volatile
continue	float	register	switch	while
default				

Tipos de datos elementales

Tipo	Significado	Tamaño en bytes
`char`	caracter	1
`int`	entero	2 – 4
`short`	entero corto	2
`long`	entero largo	4
`unsigned char`	caracter sin signo	1
`unsigned`	entero sin signo	2 – 4
`unsigned char`	entero corto sin signo	2
`unsigned long`	entero largo sin signo	4
`float`	coma flotante, real	4
`double`	coma flotante largo	8

- En UNIX todos los `int` son `long`.

Constantes

- Caracteres: **'a'**, **'b'**
- Valores enteros:
  - Notación decimal: 987
  - Notación hexadecimal: **0x**25 ó **0X**25
  - Notación octal: **0**34
  - Enteros sin signo: 485**U**
  - Enteros de tipo long: 485**L**
  - Enteros sin signo de tipo long: 485**UL**
  - Valores negativos (signo menos): -987
- Valores reales (coma flotante):
  - Ejemplos: **12**, **14**, **8.**, **.34**
  - Notación exponencial: **.2e+9**, **1.04E-12**
  - Valores negativos (signo menos): **-12 -2e+9**

La función `printf`

- Permite imprimir información por la salida estándar.
- Formato:

```
printf(formato, argumentos);
```

- Ejemplos:

```
printf("Hola mundo\n");
printf("El numero 28 es %d\n", 28);
printf("Imprimir %c %d %f\n", á', 28, 3.0e+8);
```

- Especificadores de formato:

Carácter	Argumentos	Resultado
`d, i`	entero	entero decimal con signo
`u`	entero	entero decimal sin signo
`o`	entero	entero octal sin signo
`x,X`	entero	entero hexadecimal sin signo
`f`	real	real con punto y con signo
`e, E`	real	notación exponencial con signo
`g, G`		
`c`	caracter	un caracter
`s`	cadena de caracteres	cadena de caracteres
`%`		imprime %
`p`	void	Dependiente implementación
`ld,lu,lx,lo`	entero	entero largo

La función `printf` (II)

- Secuencias de escape:

Secuencia	Significado
`\n`	nueva línea
`\t`	tabulador
`\b`	backspace
`\r`	retorno de carro
`\"`	comillas
`\'`	apóstrofo
`\\`	backslash
`\?`	signo de interrogación

- Especificadores de ancho de campo:

```
printf("Numero entero = %5d \n", 28);
```

produce la salida:

```
Numero entero =     28
```

```
printf("Numero real = %5.4f \n", 28.2);
```

produce la salida:

```
Numero entero =     28.2000
```

Variables

- Identificador utilizado para representar un cierto tipo de información.
- Cada variable es de un tipo de datos determinado.
- Una variable puede almacenar diferentes valores en distintas partes del programa.
- Debe comenzar con una letra o el caracter _
- El resto sólo puede contener letras, números o _
- Ejemplos de variables válidas:

```
numero
_color
identificador_1
```

- C es sensible a mayúsculas y minúsculas. Las siguientes variables son todas distintas:

```
pi   PI  Pi  pI
```


Declaración de variables

- Una declaración asocia un tipo de datos determinado a una o más variables.
- El formato de una declaración es:

```
tipo_de_datos  var1, var2, ..., varN;
```

- Ejemplos:

```
int a, b, c;
float numero_1, numero_2;
char letra;
unsigned long entero;
```

- Deben declararse todas las variables antes de su uso.
- Deben asignarse a las variables nombres significativos.

```
int temperatura;
int k;
```


Expresiones y sentencias

- Una **expresión** representa una unidad de datos simple, tal como un número o carácter.
- También puede estar formados por identificadores y operadores:

```
a + b
num1 * num2
```

- Una **sentencia** controla el flujo de ejecución de un programa.
  - Sentencia simple:

```
temperatura = 4 + 5;
```

  - Sentencia compuesta:

```
{
   temperatura_1 = 4 + 5;
   temperatura_2 = 8 + 9;
}
```


Sentencia de asignación

- El operador de asignación (=) asigna un valor a una variable.
- Puede asignarse valor inicial a una variable en su declaración.

```
#include <stdio.h>
main()
{
   int a = 1;
   float b = 4.0;
   int c, d;
   char letra;

   c = 10;
   letra = á';
   d = a + c;

   printf("a = %d \n", a);
   printf("b = %f \n", b);
   printf("c = %d \n", c);
   printf("d = %d \n", d);
   printf("La letra es %c \n", letra);
}
```

Función `scanf()`

- Permite leer datos del usuario.
- La función devuelve el número de datos que se han leído correctamente.
- Formato:

```
scanf(formato, argumentos);
```

- Especificadores de formato igual que `printf()`.
- Ejemplos:

```
scanf("%f", &numero);

scanf("%c", &letra);

scanf("%f %d %c", &real, &entero, &letra);

scanf("%ld",  &entero_largo);
```

- Importante el símbolo `&`.

Ejemplo

- Programa que lee un número entero y lo eleva al cuadrado:

```
#include <stdio.h>

main()
{
   int numero;
   int cuadrado;

   printf("Introduzca un numero:");
   scanf("%d", &numero);

   cuadrado = numero * numero;

   printf("El cuadrado de %d es %d\n", numero,
            cuadrado);
}
```


Introducción a la directiva `#define`

- Permite definir constantes simbólicas en el programa.

```
#define   nombre   texto
```

- `nombre` representa un nombre simbólico que suele escribirse en mayúsculas.
- `texto` no acaba en `;`
- `nombre` es sustituido por `texto` en cualquier lugar del programa.
- Ejemplos:

```
#define PI    3.141593

#define CIERTO   1
#define FALSO    0
#define AMIGA    "Marta"
```


Ejemplo

- Programa que lee el radio de un circulo y calcula su área.

```
#include <stdio.h>
#define PI    3.141593

main()
{
   float radio;
   float area;

   printf("Introduzca el radio: ");
   scanf("%f", &radio);

   area = PI * radio * radio;
   printf("El area del circulo es %5.4f \n", area);
   exit(0);
}
```

Errores de programación comunes

- Problemas con las mayúsculas y minúsculas.
- Omisión del punto y coma.
- Comentarios incompletos.
- Comentarios anidados.
- Uso de variables no declaradas.
- **Ejemplo:**

```
#include <stdio.h>
#define PI    3.141593

      /* Programa que calcula el area de
         un circulo
Main()
{
   float radio;

   printf("Introduzca el radio: ")
   scanf("%f", &radio);

   area = PI * radio * Radio;
   printf("El area del circulo es %5.4f \n", area);
}
```


OPERADORES Y EXPRESIONES

Operadores aritméticos

- Operadores aritméticos de C:

Operador	Función
$+$	suma
$-$	resta
$*$	producto
$/$	división
%	operador módulo
	resto de la división entera

- División entera: división de una cantidad entera por otra $\Longrightarrow$se desprecia la parte decimal del cociente.
- El operador % requiere que los dos operandos sean enteros.
- La mayoría de las versiones de C asignan al resto el mismo signo del primer operando.
- Valores negativos con el signo $-$. Ejemplo: -4

Ejemplos

- **Ejemplo 1**. Si $a = 10$ y $b = 3 \Longrightarrow$

Expresión	Valor
$a + b$	13
$a - b$	7
$a * b$	30
a/b	3
a % b	1

- **Ejemplo 2**. Si $a = 11$ y $b = -3 \Longrightarrow$

Expresión	Valor
$a + b$	8
$a - b$	14
$a * b$	-33
a/b	-3
a % b	2

Operadores aritméticos. Conversión de tipos

- En C un operador se puede aplicar a dos variables o expresiones distintas.
- Los operandos que difieren en tipo pueden sufrir una conversión de tipo.
- **Norma general:** El operando de menor precisión toma el tipo del operando de mayor precisión.
- **Reglas de conversión de tipos:**
  1. Si un operando es `long double` el otro se convierte a `long double`.
  2. En otro caso, si es `double` el otro se convierte a `double`.
  3. En otro caso, si es `float` el otro se convierte a `float`.
  4. En otro caso, si es `unsigned long int` el otro se se convierte a `unsigned long int`.

  5. Si un operando es `long int` y el otro es `unsigned int`, entonces:
     5.1. Si el `unsigned int` puede convertirse a `long int` el operando `unsigned int` se convertirá en `long int`.
     5.2. En otro caso, ambos operandos se convertirán a `unsigned long int`.
  6. En otro caso, si un operando es `long int` el otro se convertirá a `long int`.
  7. En otro caso, si un operando es `unsigned int` el otro se convertirá a `unsigned int`.
  8. En otro caso, ambos operandos serán convertidos a tipo `int` si es necesario.

Conversión de tipos o `cast`

- Se puede convertir una expresión a otro tipo:

```
(tipo datos) expresion
```

- **Ejemplo:**

```
( (int) 5.5 % 4)
```


Prioridad de los operadores aritméticos

- La prioridad indica el orden en el que se realizan las operaciones aritméticas.
- Las operaciones con mayor precedencia se realiza antes.

Prioridad	Operación
Primero	( )
Segundo	Negación (signo menos)
Tercero	$*, /, \%$
Cuarto	$+, -$

- Dentro de cada grupo las operaciones se realizan de izquierda a derecha.
- **Ejemplo 1:**

```
a - b / c * d
```

es equivalente a

```
a - ((b/c) * d)
```

- **Ejemplo2:**

```
(a - b) / c * (a  + c)
(a - b) / (c * d)
```


Operadores de incremento y decremento

- Operador de **incremento** $++$ incrementa en uno el operando.
- Operador de **decremento** $--$ decrementa en uno el operando.
- Variantes:
  - Postincremento $\Longrightarrow \quad i++$
  - Preincremento $\Longrightarrow \quad ++i$
  - Postdecremento $\Longrightarrow \quad i--$
  - Predecremento $\Longrightarrow \quad --i$
- **Ejemplos:**
  - La expresión $i++; \iff i = i + 1$
  - La expresión $++i; \iff i = i + 1$
  - La expresión $i--; \iff i = i - 1$
  - La expresión $--i; \iff i = i - 1$
- Si el operador *precede* al operando el valor del operando se modificará **antes** de su utilización.
- Si el operador *sigue* al operando el valor del operando se modificará **después** de su utilización.

- **Ejemplos:** si $a = 1 \Longrightarrow$

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

Imprime:

```
a = 1
a = 2
a = 2
a = 3
```

- Mayor prioridad que los operadores aritméticos.
- Ejemplo: ¿Qué hace cada uno de los siguientes fragmentos de código?

```
j = 2;              j = 2;
k = j++  + 4;       k = ++j  + 4;
```

- Evitar el uso de este tipo de sentencias.

Operadores relacionales y lógicos

- Operadores relaciones en C:

Operador	Función
$<$	menor que
$>$	mayor que
$<=$	menor o igual que
$>=$	mayor o igual que
$==$	igual que
$!=$	distinto que

- Se utilizan para formar expresiones lógicas.
- El resultado es un valor entero que puede ser:
  - *cierto* se representa con un 1
  - *falso* se representa con un 0
- **Ejemplo 1:** Si $a = 1$ y $b = 2 \Longrightarrow$

Expresión	Valor	Interpretación
$a < b$	1	cierto
$a > b$	0	falso
$(a + b) != 3$	0	falso
$a == b$	0	falso
$a == 1$	1	cierto

Prioridad de los operadores relacionales

Prioridad	Operación
Primero	( )
Segundo	$>$ $<$ $>=$ $<=$
Tercero	$==$ $!=$

- Los operadores aritméticos tienen mayor prioridad que los operadores relacionales.
- Dentro de cada grupo las operaciones se realizan de izquierda a derecha.
- **Ejemplo:** Si $a = 1$ y $b = 2 \Longrightarrow$

Expresión	Valor	Interpretación
$a < b == a > b$	0	falso
$(a < b) == (a > b)$	0	falso
$1 > (a == 2)$	1	cierto
$a == (b == 2)$	1	cierto

Operadores lógicos

- Operadores lógicos en C:

Operador	Función
&&	Y *lógica* (AND)
||	O *lógica* (OR)
!	NO *lógico* (NOT)

- Actúan sobre operandos que son a su vez expresiones lógicas que se interpretan como:
  - *cierto*, cualquier valor distinto de 0
  - *falso*, el valor 0
- Se utilizan para formar expresiones lógicas.
- El resultado es un valor entero que puede ser:
  - *cierto* se representa con un 1
  - *falso* se representa con un 0

Tablas de verdad de los operadores lógicos

- Indican el resultado de los operadores lógicos.
- Tabla de verdad del Y *lógico*:

Expresión	Valor	Interpretación
1 && 1	1	cierto
1 && 0	0	falso
0 && 1	0	falso
0 && 0	0	falso

- Tabla de verdad del O *lógico*:

Expresión	Valor	Interpretación
1 || 1	1	cierto
1 || 0	1	cierto
0 || 1	1	cierto
0 || 0	0	falso

- Tabla de verdad de la negación *lógica*:

Expresión	Valor	Interpretación
!1	0	falso
!0	1	cierto

Prioridad de los operadores lógicos

Prioridad	Operación
Primero	( )
Segundo	!
Tercero	&&
Cuarto	||

- && y || se evalúan de izquierda a derecha.
- ! se evalúa de derecha a izquierda.
- **Ejemplos:** si $a = 7$ y $b = 3 \Longrightarrow$

Expresión	Valor	Interpretación
$(a + b) < 10$	0	falso
$!((a + b) < 10)$	1	cierto
$(a\ != 2)\ ||\ ((a + b)\ <= 10)$	1	cierto
$(a > 4)\ \&\&\ (b < 5)$	1	cierto

- Las expresiones lógicas con && o || se evalúan de izquierda a derecha, **sólo** hasta que se ha establecido su valor cierto/falso.
- **Ejemplo:** si $a = 8$ y $b = 3 \Longrightarrow$ en la sentencia

```
a > 10  &&  b < 4
```

no se evaluará $b < 4$ puesto que $a > 10$ es falso, y por lo tanto el resultado final será falso.

- **Ejemplo:** si $a = 8$ y $b = 3 \Longrightarrow$ en la sentencia

```
a < 10  ||  b < 4
```

no se evaluará $b < 4$ puesto que $a < 10$ es verdadero, y por lo tanto el resultado final será verdadero.

Resumen de prioridades

- Relación de las prioridades de los distintos operadores (vistos hasta ahora) de mayor a menor prioridad.

Operador	Evaluación
$++$ $--$ (*post*)	I →D
$++$ $--$ (*pre*)	D →I
! $-$ (signo menos)	D →I
(*tipo*) (*cast*)	D →I
$*$ / %	I →D
$+$ $-$	I →D
$<$ $>$ $<=$ $>=$	I →D
$==$ $!=$	I →D
&&	I →D
||	I →D

- **Ejemplos:** si $a = 7$ y $b = 3 \Longrightarrow$

Expresión	Valor	Interpretación
$a + b < 10$	0	falso
$a\ != 2\ ||\ a + b\ <= 10$	1	cierto
$a > 4\ \&\&\ b < 5$	1	cierto

Operadores de asignación

- Forma general:

```
identificador = expresion
```

- **Ejemplos:**

```
a = 3;
area = lado * lado;
```

- El operador de asignación = y el el igualdad == son **distintos** $\Longrightarrow$ERROR COMÚN.
- **Asignaciones múltiples:**

```
id_1 = id_2 = ... = expresion
```

- Las asignaciones se efectúan de derecha a izquierda.
- En `i = j = 5`
  1. A j se le asigna 5
  2. A i se le asigna el valor de j

Reglas de asignación

- Si los dos operandos en una sentencia de asignación son de tipos distintos, entonces el valor del operando de la derecha será automáticamente convertido al tipo del operando de la izquierda. Además:
  1. Un valor en coma flotante se puede truncar si se asigna a una variable de tipo entero.
  2. Un valor de doble precisión puede redondearse si se asigna a una variable de coma flotante de simple precisión.
  3. Una cantidad entera puede alterarse si se asigna a una variable de tipo entero más corto o a una variable de tipo carácter.
- Es **importante** en C utilizar de forma correcta la conversión de tipos.

Operadores de asignación compuestos

- `+=  -=  *=  /=  %=`

Expresión	Expresión equivalente
`j += 5`	`j = j + 5`
`j -= 5`	`j = j - 5`
`j *= 5`	`j = j * 5`
`j /= 5`	`j = j / 5`
`j %= 5`	`j = j % 5`

- Los operadores de asignación tienen menor prioridad que el resto.

Ejemplo

- Programa que convierte grados Fahrenheit a grados centígrados.

$$C = (5/9) * (F - 32)$$

```
#include <stdio.h>

main()
{
  float centigrados;
  float fahrenheit;

  printf("Introduzca una temperatura en grados
          fahrenheit: ");
  scanf("%f", &fahrenheit);

  centigrados = 5.0/9 * (fahrenheit - 32);

  printf("%f grados fahrenheit = %f grados
      centigrados \n", fahrenheit, centigrados);
}
```

- Salida del programa:

```
Introduzca una temperatura en grados fahrenheit: 96
96.0000 grados fahrenheit = 35.5557 grados centigrados
```

- ¿Qué ocurriría si se hubiera utilizado la siguiente sentencia?

```
centigrados = 5/9 * (fahrenheit - 32);
```


Operador condicional

- Forma general:

```
expresion_1 ? expresion_2 : expresion_3
```

- Si `expresion_1` es *verdadero* $\Longrightarrow$devuelve `expresion_2`
- Si `expresion_1` es *falso* $\Longrightarrow$devuelve `expresion_3`
- Su prioridad es justamente superior a los operadores de asignación.
- Se evalúa de derecha a izquierda.
- **Ejemplo:** si $a = 1$ $\Longrightarrow$en la sentencia:

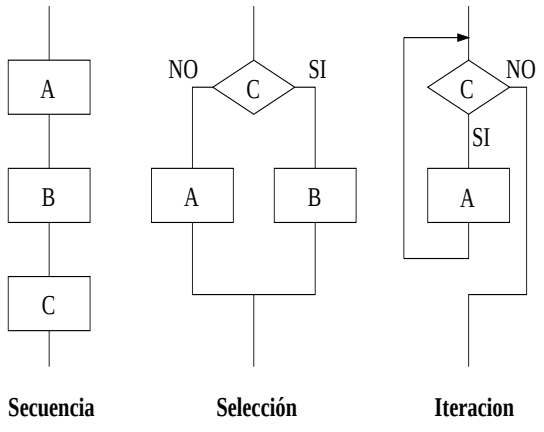
```
k = (a < 0) ? 0 : 100;
```

  - Primero se evalúa `(a < 0)`
  - Como es *falso* el operador condicional devuelve `100`
  - Este valor se asigna a `k`. Es decir `k` toma el valor `100`

SENTENCIAS DE CONTROL

Introducción

- Tipos de estructuras de programación:
  - **Secuencia**: ejecución sucesiva de dos o más operaciones.
  - **Selección**: se realiza una u otra operación, dependiendo de una condición.
  - **Iteración**: repetición de una operación mientras se cumpla una condición.



Sentencia `if`

- Forma general:

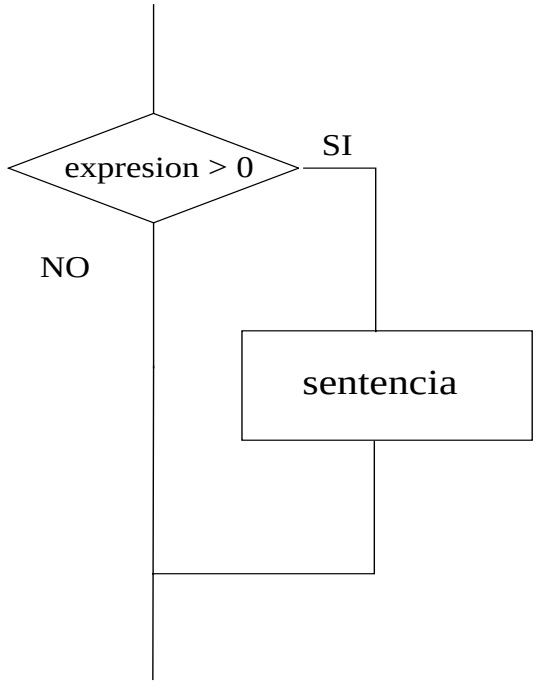
```
if (expresion)
   sentencia
```

- si `expresion` es verdadera (valor mayor que 0) $\Longrightarrow$se ejecuta `sentencia`.
- La `expresion` debe estar entre paréntesis.
- Si `sentencia` es compuesta $\Longrightarrow$

```
if (expresion)
{
   sentencia 1
   sentencia 2
       .
       .
       .
   sentencia N
}
```


Sentencia `if`

- Diagrama de flujo:



Ejemplo 1

- Programa que lee un número e indica si es par.

```
#include <stdio.h>

main()
{
   int numero;

   /* leer el numero */
   printf("Introduzca un numero: ");
   scanf("%d", &numero);

   if ((numero % 2) == 0)
       printf("El numero %d es par.\n", numero);
}
```


Ejemplo 2

- Programa que lee un número y lo eleva al cuadrado si es par.

```
#include <stdio.h>

main()
{
   int numero;
   int cuadrado;

   /* leer el numero */
   printf("Introduzca un numero: ");
   scanf("%d", &numero);

   if ((numero % 2) == 0)
   {
       cuadrado = numero * numero;
       printf("El cuadrado de %d es %d.\n", numero,
                 cuadrado);
   }
}
```

Sentencia `if-else`

- Forma general:

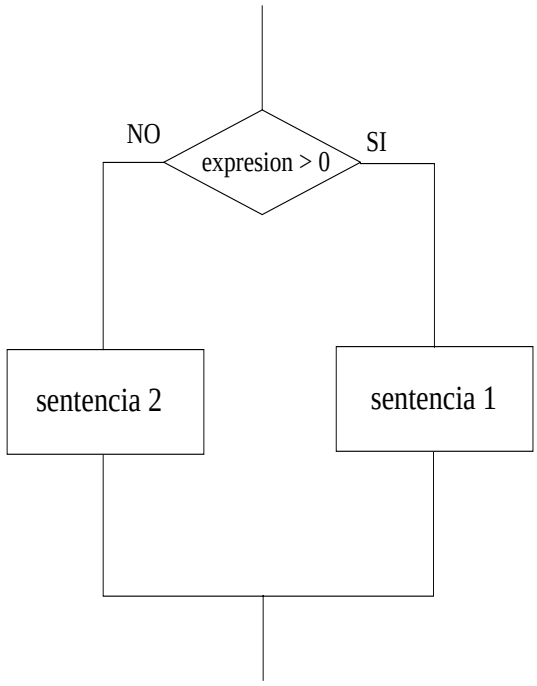
```
if (expresion)
   sentencia 1
else
   sentencia 2
```

- si `expresion` es:
  - verdadera (valor mayor que 0) $\Longrightarrow$se ejecuta `sentencia 1`.
  - falsa (valor igual a 0) $\Longrightarrow$se ejecuta `sentencia 2`.
- Si las sentencias son compuestas se encierran entre `{ }`
- Las sentencias pueden ser a su vez sentencias `if-else`

```
if (e1)
   if (e2)
      S1
   else
      S2
else
   S3
```


Sentencia `if-else`

- Diagrama de flujo:



Ejemplo

- Programa que lee un número y dice si es par o impar.

```
#include <stdio.h>

main()
{
    int numero;

   /* Leer el numero */
   printf("Introduzca un numero: ");
   scanf("%d", &numero);

   if ((numero % 2) == 0)
      printf("El numero %d es par.\n", numero);
   else
      printf("El numero %d es impar.\n", numero);

}
```


Sentencia `for`

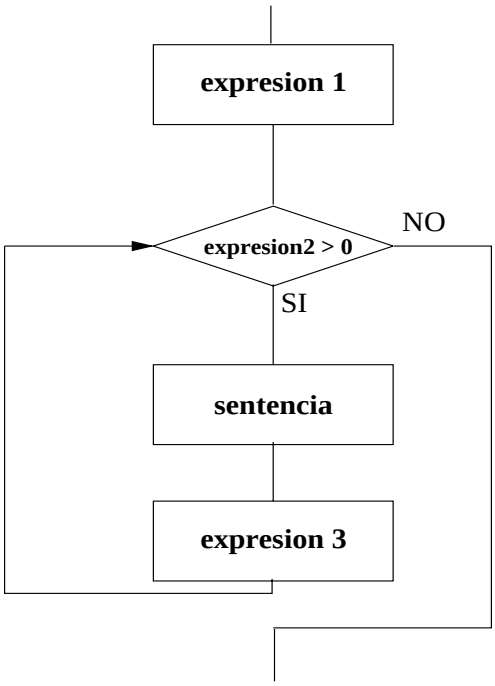
- Forma general:

```
for (expresion 1; expresion 2; expresion 3)
   sentencia
```

- Inicialmente se ejecuta `expresion 1` $\Longrightarrow$se inicializa algún parámetro que controla la repetición del bucle.
- `expresion 2` es una condición que debe ser cierta para que se ejecute `sentencia`.
- `expresion 3` se utiliza para modificar el valor del parámetro.
- El bucle se repite mientras `expresion 2` no sea cero (falso).
- Si `sentencia` es compuesta se encierra entre `{ }`
- `expresion 1` y `expresion 3` se pueden omitir.
- Si se omite `expresion 2` se asumirá el valor permanente de 1 (cierto) y el bucle se ejecutará de forma indefinida.

Sentencia `for`

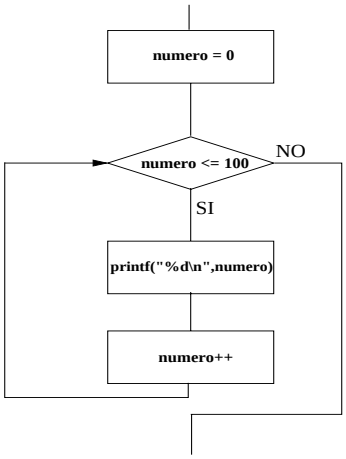
- Diagrama de flujo:



Ejemplo

- Programa que imprime los 100 primeros números

```
#include <stdio.h>

main()
{
    int numero;

    for (numero=0; numero <100; numero++)
       printf("%d\n", numero);
}
```



Sentencia `while`

- Forma general:

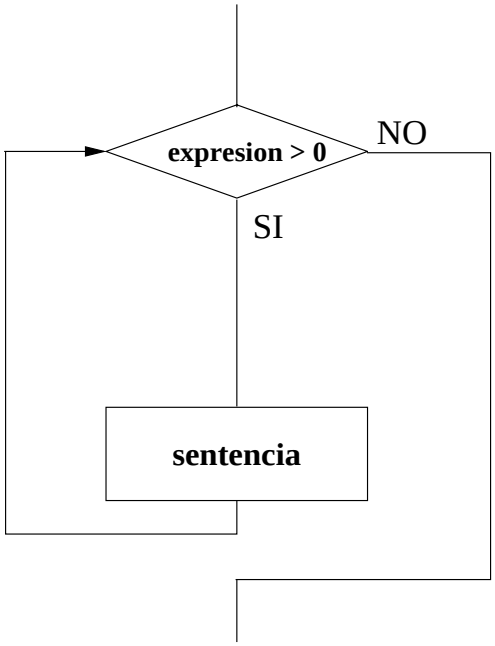
```
while (expresion)
   sentencia
```

- `sentencia` se ejecutará mientras el valor de `expresion` sea verdadero (distinto de 0).
- Primero se evalúa `expresion`.
- Lo normal es que `sentencia` incluya algún elemento que altere el valor de `expresion`, proporcionando así la condición de salida del bucle.
- Si la sentencia es compuesta se encierra entre `{ }`

```
while (expresion)
{
   sentencia 1
   sentencia 2
        .
        .
   sentencia N
}
```


Sentencia `while`

- Diagrama de flujo:



Ejemplo

- Programa que lee un número N y calcula $1 + 2 + 3 + \ldots + N$

```
#include <stdio.h>

main()
{
   int N;
   int suma = 0;

   /* leer el numero N */
   printf("N: ");
   scanf("%d", &N);

   while (N > 0)
   {
      suma = suma + N;
      N = N - 1;      /* equivalente a N-- */
   }

   printf("1 + 2 +...+ N = %d\n", suma);
}
```


Sentencia `do-while`

- Formal general:

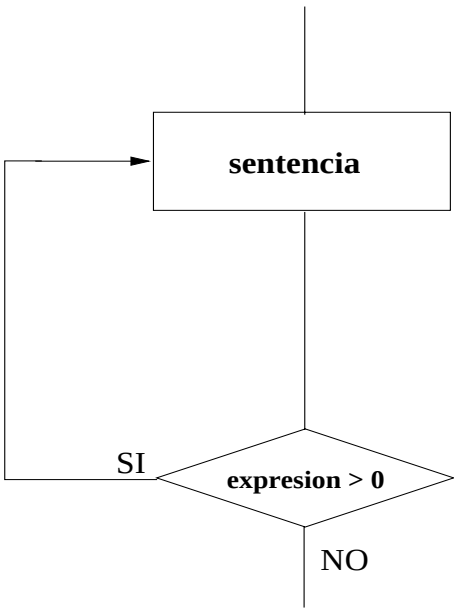
```
do
   sentencia
while (expresion);
```

- `sentencia` se ejecutará mientras el valor de `expresion` sea verdadero (distinto de 0).
- `sentencia` siempre se ejecuta al menos una vez (diferente a `while`).
- Lo normal es que `sentencia` incluya algún elemento que altere el valor de `expresion`, proporcionando así la condición de salida del bucle.
- Si la sentencia es compuesta se encierra entre `{ }`

```
do
{
   sentencia 1
   sentencia 2
       .
       .
   sentencia N
}while (expresion);
```

- Para la mayoría de las aplicaciones es mejor y más natural comprobar la condición antes de ejecutar el bucle (bucle `while`).

- Diagrama de flujo:



Ejemplo

- Programa que lee de forma repetida un número e indica si es par o impar. El programa se repite mientras el numero sea distinto de cero.

```
#include <stdio.h>

main()
{
   int numero;

   do
   {
      /* se lee el numero */
      printf("Introduzca un numero: ");
      scanf("%d", &numero);

      if ((numero % 2) == 0)
         printf("El numero %d es par.\n", numero);
      else
         printf("El numero %d es par.\n", numero);

   } while (numero != 0)
}
```

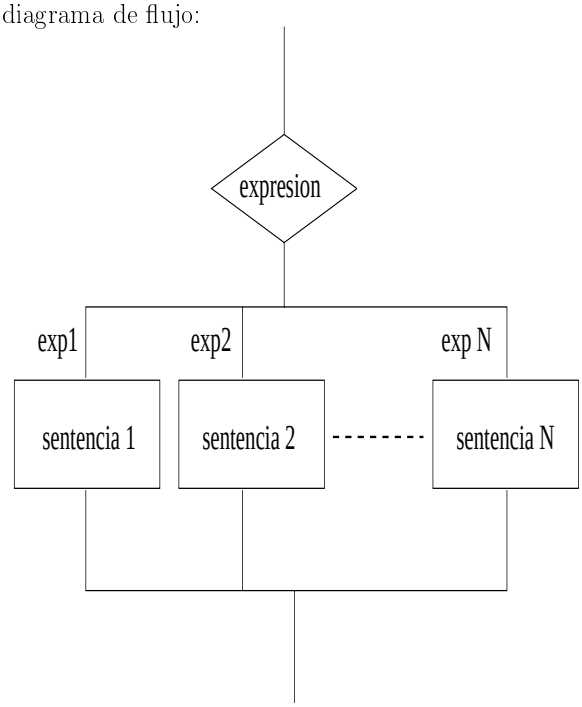
Sentencia `switch`

- Formal general:

```
switch (expresion)
{
   case exp 1:
      sentencia 1;
      sentencia 2;
         .
         .
      break;

   case exp N:
   case exp M:
      sentencia N1;
      sentencia N2;
         .
         .
      break;

   default:
      sentencia D;
         .
}
```

- `expresion` devuelve un valor entero (también puede ser de tipo `char`).

- `exp 1`, ..., `exp N` representan expresiones constantes de valores enteros (también caracteres).
- diagrama de flujo:



Ejemplo

```
#include <stdio.h>

main()
{
   char letra;

   printf("Introduzca una letra: ");
   scanf("%d", &letra);

   switch(letra)
   {
      case á':
      case Á':
         printf("Vocal %c\n", letra);
         break;
      case é':
      case É':
         printf("Vocal %c\n", letra);
         break;
      case í':
      case Í':
         printf("Vocal %c\n", letra);
         break;
      case ó':
      case Ú':
         printf("Vocal %c\n", letra);
         break;
      case ú':
      case Ú':
```



```
         printf("Vocal %c\n", letra);
         break;
      default:
         printf("Consonante %c\n", letra);
   }
}
```

Bucles anidados

- Los bucles se pueden *anidar* unos en otros.
- Se pueden anidar diferentes tipos de bucles.
- Importante estructurarlos de forma correcta.
- **Ejemplo:** Calcular $1 + 2 + \ldots N$ mientras N sea distinto de 0.

```
#include <stdio.h>
main()
{
   int N;
   int suma;
   int j;

   do
   {
      /* leer el numero N */
      printf("Introduzca N: ");
      scanf("%d", &N);

      suma = 0;
      for (j = 0; j <= N; j++) /* bucle anidado */
         suma = suma + j;
      printf("1 + 2 + ... + N = %d\n", suma);
   } while (N > 0);     /* fin del bucle do */
}
```


Sentencia `break`

- Se utiliza para terminar la ejecución de bucles o salir de una sentencia `switch`.
- Es necesaria en la sentencia `switch` para transferir el control fuera de la misma.
- En caso de bucles anidados, el control se transfiere fuera de la sentencia más interna en la que se encuentre, pero no fuera de las externas.
- **No** es aconsejable el uso de esta sentencia en bucles $\Longrightarrow$contrario a la programación estructurada.
- Puede ser útil cuando se detectan errores o condiciones anormales.
- Ejemplo: programa que calcula la media de un conjunto de datos.
  - Primera versión: utiliza la sentencia `break`.
  - Segunda versión: estructurada $\Longrightarrow$no utiliza la sentencia `break`.

Versión con `break`

```
#include <stdio.h>

#define TRUE    1
#define FALSE   0

main()
{
   float valor;
   float suma = 0.0;
   float media;
   int cont;
   int total;
   int error = FALSE;
   int leidos;

   printf("Cuantos datos: ");
   scanf("%d", &total);

   for(cont = 0; cont < total; cont++)
   {
      printf("Introduzca valor: ");
      leidos = scanf("%f", &valor);
      if (leidos == 0)
      {
         error = TRUE;
         break;
      }

      suma = suma + valor;
   }
```



```
   if (error)
      printf("Error en la lectura de los datos\n");
   else
   {
      if (total > 0)
      {
         media = suma / total;
         printf("Media = %f\n", media);
      }
      else
         printf("El numero de datos es 0\n");
   }
}
```

Versión estructurada

```
#include <stdio.h>

#define TRUE    1
#define FALSE   0

main()
{
   float valor;
   float suma = 0.0;
   float media;
   int cont = 0;
   int total;
   int error = FALSE;
   int leidos;

   printf("Cuantos datos: ");
   scanf("%d", &total);

   while ((cont < total) && (!error))
   {
      printf("Introduzca valor: ");
      leidos = scanf("%f", &valor);
      if (leidos == 0)
         error = TRUE;
      else
      {
         suma = suma + valor;
         cont = cont + 1;
      }
   }
```



```
   if (error)
      printf("Error en la lectura de los datos\n");
   else
   {
      if (total > 0)
      {
         media = suma / total;
         printf("Media = %f\n", media);
      }
      else
         printf("El numero de datos es 0\n");
   }
}
```


FUNCIONES Y PROGRAMACIÓN ESTRUCTURADA

Introducción

- Una función es un segmento de programa que realiza una determinada tarea.
- Todo programa C consta de una o más funciones.
- Una de estas funciones se debe llamar `main()`.
- Todo programa comienza su ejecución en la función `main()`.
- El uso de funciones permite la descomposición y desarrollo modular. Permite dividir un programa en componentes más pequeños: funciones.

Ejemplo

- Programa que calcula el máximo de dos números.

```
#include <stdio.h>
int maximo(int a, int b);   /* prototipo de funcion */

main()
{
    int x, y;
    int max;

    printf("Introduzca dos numeros: ");
    scanf("%d %d", &x, &y);

    max = maximo(x,y);    /* llamada a la funcion */
    printf("El maximo es %d\n", max);
}

int maximo(int a, int b) /* definicion de la funcion */
{
    int max;

    if (a > b)
        max = a;
    else
        max = b;
    return(max); /* devuelve el valor maximo */
}
```


Ejemplo (II)

- Programa que dice si un número es cuadrado un perfecto.

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

#define TRUE     1
#define FALSE    0

void explicacion(void);
int cuadrado_perfecto(int x);

main()
{
   int n;
   int perfecto;

   explicacion();
   scanf("%d", &n);
   perfecto = cuadrado_perfecto(n);
   if (perfecto)
      printf("%d es cuadrado perfecto.\n", n);
   else
      printf("%d no es cuadrado perfecto.\n", n);
}
```



```
void explicacion(void)
{
   printf("Este programa dice si un numero ");
   printf("es cuadrado perfecto \n");
   printf("Introduzca un numero: );
}

int cuadrado_perfecto(int x)
{
   int raiz;
   int perfecto;

   raiz = (int) sqrt(x);
   if (x == raiz * raiz)
      perfecto = TRUE;   /* cuadrado perfecto */
   else
      perfecto = FALSE; /* no es cuadrado perfecto */

   return(perfecto);
}
```

- Para compilar este programa:

```
gcc -Wall -o perfecto perfecto.c -lm
```


Definición de una función

```
tipo nombre(tipo1 arg1, ..., tipoN argN)
{
    /* CUERPO DE LA FUNCION */
}
```

- Los argumentos se denominan **parámetros formales**.
- La función devuelve un valor de tipo `tipo`.
- Si se omite `tipo` se considera que devuelve un `int`.
- Si no devuelve ningún tipo $\Longrightarrow$`void`.
- Si no tiene argumentos $\Longrightarrow$`void`.

```
void explicacion(void)
```

- Entre llaves se encuentra el *cuerpo* de la función (igual que `main()`).
- La sentencia `return` finaliza la ejecución y devuelve un valor a la función que realizó la llamada.

```
return(expresion);
```

- Una función sólo puede devolver un valor.
- En C no se pueden anidar funciones.

Declaración de funciones: prototipos

```
tipo nombre(tipo1 arg1, ..., tipoN argN);
```

- No es obligatorio pero si aconsejable.
- Permite la comprobación de errores entre las llamadas a una función y la definición de la función correspondiente.
- Ejemplo: función que calcula x elevado a y (con y entero).

```
float potencia (float x, int y); /* prototipo */


float potencia (float x, int y)  /* definicion */
{
   int i;
   float prod = 1;

   for (i = 0; i < y; i++)
      prod = prod * x;

   return(prod);
}
```


Llamadas a funciones

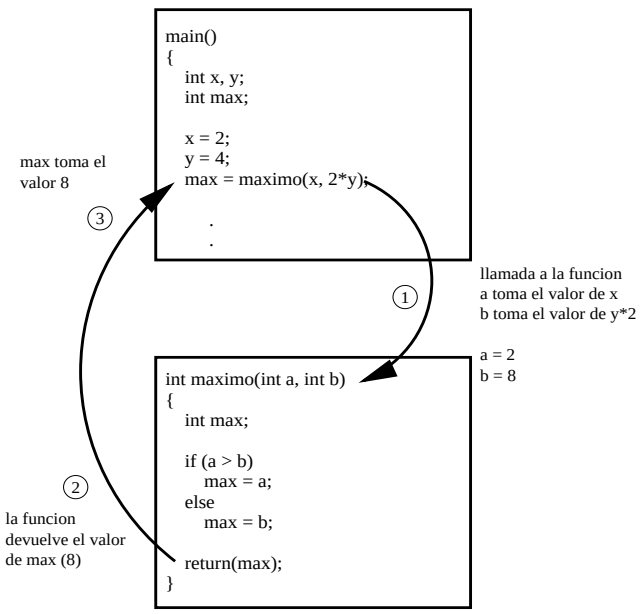
- Para llamar a una función se especifica su nombre y la lista de argumentos.

```
maximo(2, 3);
```

- **Parámetros formales**: los que aparecen en la definición de la función.
- **Parámetros reales**: los que se pasan en la llamada a la función.
- En una llamada habrá un argumento real por cada argumento formal.
- Los parámetros reales pueden ser:
  - Constantes.
  - Variables simples.
  - Expresiones complejas

  Pero deben ser del mismo tipo de datos que el argumento formal correspondiente.
- Cuando se pasa un valor a una función se copia el argumento real en el argumento formal.

- Se puede modificar el argumento formal dentro de la función, pero el valor del argumento real no cambia: **paso de argumentos por valor**.
- Proceso de llamada:



Ejemplo

- Programa que lee el número de caracteres introducidos hasta fin de fichero.

```
#include <stdio.h>
int cuenta_caracteres(void);

main()
{
   int num_car;

   num_car = cuenta_caracteres();
   printf("Hay %d caracteres\n", num_car);
}

int cuenta_caracteres(void)
{
   char c;
   int cont = 0;

   c = getchar();
   while (c != EOF)
   {
      cont = cont + 1;
      c = getchar();
   }
   return(cont);
}
```

Recursividad

- Una función se llama a sí misma de forma repetida hasta que se cumpla alguna condición.
- Ejemplo: el factorial de un número:

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * f(n-1) & \text{si } n > 0 \end{cases}$$

- Función que calcula el factorial de forma recursiva:

```
long int factorial(int n)
{
   if (n <= 1)
      return(1);
   else
      return(n * factorial(n-1));
}
```


Macros

- La sentencia `#define` se puede utilizar para definir macros.
- Una **macro** es un identificador equivalente a una expresión, sentencia o grupo de sentencias.
- Ejemplo: programa que calcula el máximo de dos números.

```
#include <stdio.h>

#define maximo(a,b)     ((a > b) ? a : b)

main()
{
    int x, y;
    int max;

    printf("Introduzca dos numeros: ");
    scanf("%d %d", &x, &y);

    max = maximo(x,y);    /* uso de la macro */
    printf("El maximo es %d\n", max);
}
```


- No puede haber blancos entre el identificador y el paréntesis izquierdo.
- Una macro no es una llamada a función.
- El preprocesador sustituye todas las referencias a la macro que aparezcan dentro de un programa antes de realizar la compilación:

```
main()
{
    int x, y;
    int max;

    printf("Introduzca dos numeros: ");
    scanf("%d %d", &x, &y);

    max = ((x > y) ? x : y);
    printf("El maximo es %d\n", max);
}
```

- No se produce llamada a función $\Longrightarrow$mayor velocidad.
- Se repite el código en cada uso de la macro $\Longrightarrow$mayor código objeto.

Ejemplo de macro (II)

- Dada la siguiente definición de macro:

```
#define maximo (x,y,z)  if (x > y)     \
                           z = x;  \
                        else           \
                           z = y;
```

- Cuando el preprocesador encuentra:

```
maximo(a, b, max);
```

- Lo sustituiría por:

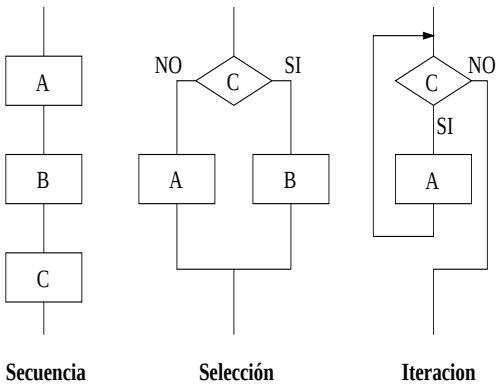
```
if ( a  >   b ) max =  a ; else  max =  b ; ;
```

- Que es equivalente a:

```
if (a > b)
   max = a;
else
   max = b;
```

Programación estructurada

- Tiende a construir programas fácilmente comprensibles.
- Se basa en la técnica de diseño mediante refinamiento progresivo: las operaciones se van descomponiendo poco a poco hasta llegar a operaciones básicas.
- Construcciones básicas de la programación estructurada:



- Importante:
  - Todos los bloques y funciones tienen un único punto de entrada.
  - Todos los bloques y funciones tienen un único punto de salida.

Ejemplo

- Programa que calcula la hipotenusa de un triángulo rectángulo.

$$h = \sqrt{(a^2 + b^2)}$$

- Pasos a seguir:
  1. Leer a y b $\Longrightarrow$función `scanf()`
  2. Calcular h según la fórmula dada $\Longrightarrow$definimos una función `hipotenusa()`.
  3. Imprimir el valor de h $\Longrightarrow$función `printf()`.

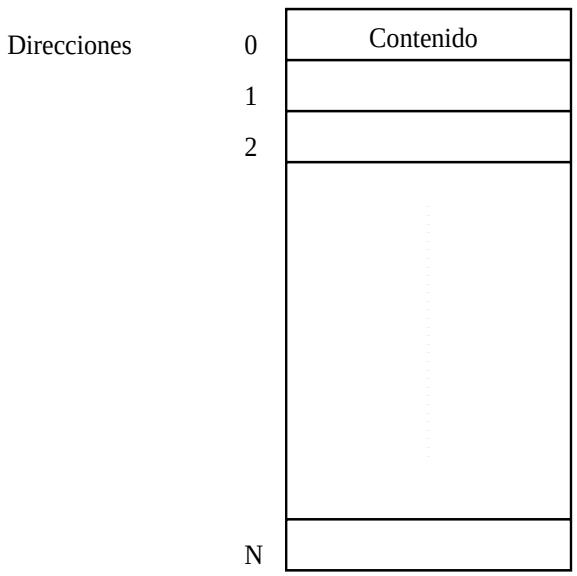

```
#include <stdio.h>
#include <math.h>
float hipotenusa(float a, float b);  /* prototipo */

main()
{
    float a, b;
    float h;
    int error;

    printf("Introduzca a y b: ");
    error = scanf("%f %f", &a, &b);
    if (error !=2)
       printf("Error al leer a y b\n");
    else
    {
        h = hipotenusa(a,b);
        printf("La hipotenusa es %f\n", h);
    }
}
float hipotenusa(float a, float b)
{
   float h;

   h = sqrt(pow(a,2) + pow(b, 2));
   return(h);
}
```

PUNTEROS Y ÁMBITO DE LAS VARIABLES

Introducción

- La memoria del computador se encuentra organizada en grupos de **bytes** que se denominan **palabras**.
- Dentro de la memoria cada dato ocupa un número determinado de bytes:
  - Un `char` $\Longrightarrow$1 byte.
  - Un `int` $\Longrightarrow$4 bytes.
- A cada byte o palabra se accede por su dirección.
- Si `x` es una variable que representa un determinado dato el compilador reservará los bytes necesarios para representar `x` (4 bytes si es de tipo `int`).

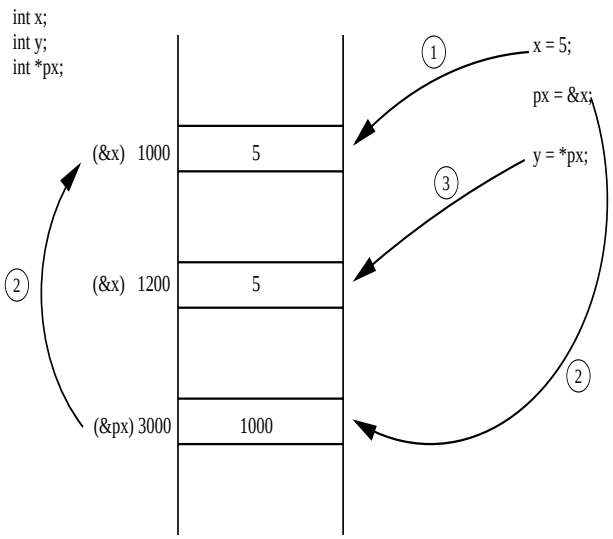
Organización de la memoria



- Si `x` es una variable $\Longrightarrow$`&x` representa la dirección de memoria de `x`.
- `&` es el **operador de dirección**.
- Un **puntero** es una variable que almacena la dirección de otro objeto (variable, función, ...).
- Ejemplo:

```
#include <stdio.h>

main()
{
   int x;    /* variable de tipo entero */
   int y;    /* variable de tipo entero */
   int *px;  /* variable de tipo
                puntero a entero */

   x = 5;
   px = &x; /* asigna a px la direccion de x */
   y = *px; /* asigna a y el contenido de la
               direccion almacenada en px */

   printf("x = %d\n", x);
   printf("y = %d\n", y);
   printf("*px = %d\n", *px);
}
```

- `*px` representa el contenido almacenado en la dirección a la que apunta `px`.
- `*` es el **operador de indirección** y opera sobre una variable de tipo puntero.
- Un puntero representa la dirección de memoria del objeto al que apunta, **NO** su valor.



Declaración de punteros

- Deben declararse antes de usarlos (igual que el resto de variables).

```
tipo_dato  *variable_ptr;
```

- `variable_ptr` es el nombre de la variable puntero.
- `tipo_dato` se refiere al tipo de dato apuntado por el puntero.
- Ejemplos:
  - `int *numero`;
  - `float *p`;
  - `char *letra`;
- `variable_ptr` sólo puede almacenar la dirección de variables de tipo `tipo_dato`.
- Para usar un puntero se debe estar seguro de que apunta a una dirección de memoria correcta.

- Un puntero no reserva memoria. El siguiente fragmento de programa es incorrecto:

```
int *p;

*p = 5;
```

- ¿A qué dirección de memoria apunta `p`?
- ¿Dónde se almacena el 5?
- El siguiente fragmento si es correcto:

```
char *p;
char letra;

letra = á';
p = &letra;
```

- `p` almacena la dirección de `letra` $\Longrightarrow$apunta a una dirección conocida.
- `*p` representa el valor almacenado en `letra` (`á'`).
- Pueden asignarse punteros del mismo tipo entre sí (ver ejemplo).

Ejemplo

- Dado el siguiente fragmento de código:

```
float n1;
float n2;
float *p1;
float *p2;

n1 = 4.0;
p1 = &n1;

p2 = p1;

n2 = *p2;

n1 = *p1 + *p2;
```

- ¿Cuánto vale `n1` y `n2`?
- `n1` = 8.0
- `n2` = 4.0;

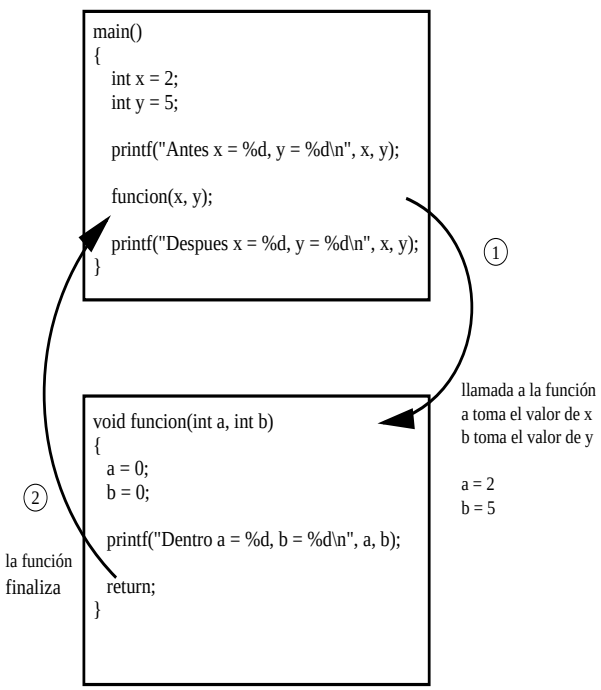
Paso de punteros a una función

- Cuando se pasa un puntero a una función **no** se pasa una copia sino la dirección del dato al que apunta.
- El uso de punteros permite el paso de argumentos por **referencia**.
- Cuando un argumento se pasa por valor, el dato se *copia* a la función.
  - Un argumento pasado por valor no se puede modificar.
- Cuando se pasa un argumento por *referencia* (cuando un puntero se pasa a una función), se pasa la *dirección* del dato $\Longrightarrow$el contenido de la dirección se puede modificar en la función.
  - Un argumento pasado por referencia si se puede modificar.
- El uso de punteros como argumentos de funciones permite que el dato sea alterado globalmente dentro de la función.

Paso de parámetros por valor (ejemplo)

```
#include <stdio.h>

void funcion(int a, int b); /* prototipo */

main()
{
   int x = 2;
   int y = 5;

   printf("Antes x = %d, y = %d\n", x, y);

   funcion(x, y);

   printf("Despues x = %d, y = %d\n", x, y);
}

void funcion(int a, int b)
{
   a = 0;
   b = 0;

   printf("Dentro a = %d, b = %d\n", a, b);

   return;
}
```

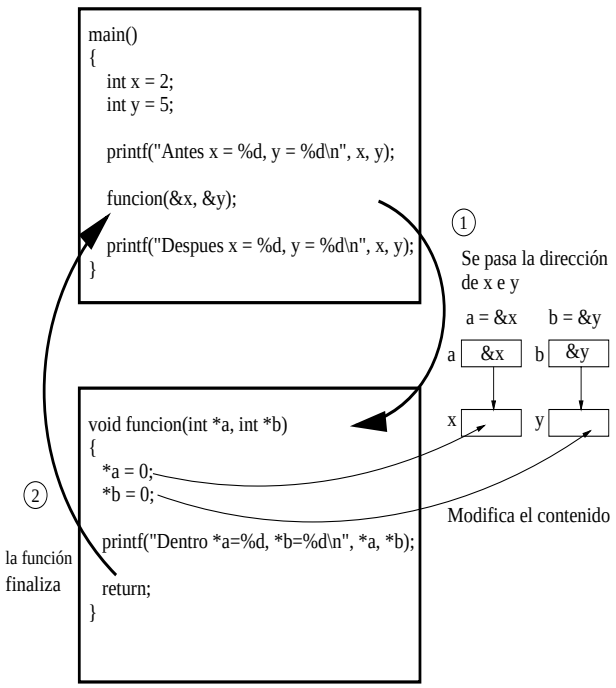

Proceso de llamada



Paso de parámetros por referencia (ejemplo)

```
#include <stdio.h>

void funcion(int *a, int *b); /* prototipo */

main()
{
   int x = 2;
   int y = 5;

   printf("Antes x = %d, y = %d\n", x, y);

   funcion(&x, &y);

   printf("Despues x = %d, y = %d\n", x, y);
}

void funcion(int *a, int *b)
{
   *a = 0;
   *b = 0;

   printf("Dentro *a = %d, *b = %d\n", *a, *b);

   return;
}
```

Proceso de llamada



Ejemplo

- Función que intercambia el valor de dos variables.

```
#include <stdio.h>

void swap(int *a, int *b); /* prototipo */

main()
{
   int x = 2;
   int y = 5;

   printf("Antes x = %d, y = %d\n", x, y);
   swap(&x, &y);
   printf("Despues x = %d, y = %d\n", x, y);
}

void swap(int *a, int *b)
{
   int temp;

   temp = *b;
   *b = *a;
   *a = temp;

   return;
}
```


Puntero NULL

- Cuando se asigna **0** a un puntero, éste no apunta a ningún objeto o función.
- La constante simbólica **NULL** definida en `stdio.h` tiene el valor **0** y representa el puntero nulo.
- Es una buena técnica de programación asegurarse de que todos los punteros toman el valor **NULL** cuando no apuntan a ningún objeto o función.

```
int *p = NULL;
```

- Para ver si un puntero no apunta a ningún objeto o función:

```
if (p == NULL)
   printf("El puntero es nulo\n");
else
   printf("El contenido de *p es\n", *p);
```


La función `sizeof`

- Devuelve el tamaño en bytes que ocupa un tipo o variable en memoria.

```
#include <stdio.h>

main()
{
  float num;

  printf(ün int ocupa %d bytes\n", sizeof(int));
  printf(ün char ocupa %d bytes\n", sizeof(char));
  printf(ün float ocupa %d bytes\n", sizeof(float));
  printf(ün double ocupa %d bytes\n", sizeof(double));
  printf("num ocupa %d bytes\n", sizeof(num));
}
```

Ámbito de las variables y tipos de almacenamiento

- Existen dos formas de caracterizar una variable:
  - Por su *tipo de datos*.
  - Por su *tipo de almacenamiento*.
- El tipo de dato se refiere al tipo de información que representa la variable (`int`, `char`, ...).
- El tipo de almacenamiento se refiere a su permanencia y a su *ámbito*.
- El **ámbito** de una variable es la porción del programa en la cual se reconoce la variable.
- Según el ámbito, las variables pueden ser:
  - Variables locales.
  - Variables globales.
- Según el tipo, las variables pueden ser:
  - Variables automáticas.
  - Variables estáticas.
  - Variables externas.
  - Variables de tipo registro.

Variables locales

- Sólo se reconocen dentro de la función donde se definen. Son invisibles al resto.
- Una variable local normalmente no conserva su valor una vez que el control del programa se transfiere fuera de la función.

```
#include <stdio.h>
void funcion1(void);

main()
{
   int a = 1;    /* variable local */
   int b = 2;    /* variable local */

   funcion1();
   printf("a = %d, b = %d \n", a, b);
}
void funcion1(void)
{
   int a = 3;    /* variable local */
   int c = 4;    /* variable local */

   printf("a = %d, c = %d \n", a, c);
   return;
}
```


Variables globales

- Se declaran fuera de las funciones y antes de su uso.
- Pueden ser accedidas desde cualquier función.

```
#include <stdio.h>
void funcion1(void);
int a = 1000;    /* variable global */

main()
{
   int b = 2;    /* variable local */

   funcion1();
   printf("a = %d, b = %d \n", a, b);
}
void funcion1(void)
{
   int c = 4;    /* variable local */

   printf("a = %d, c = %d \n", a, c);
   return;
}
```


Variables globales (II)

- Mantienen los valores que se les asignan en las funciones.
- Es mejor hacer uso de variables locales para evitar efectos secundarios o laterales.

```
#include <stdio.h>
void funcion1(void);
int a;    /* variable global */

main()
{

   printf("Antes a = %d\n", a);
   funcion1();
   printf("Despues a = %d\n", a);
}
void funcion1(void)
{
   a = 1000;
   return;
}
```

Precaución con el uso de variables globales

- El uso de variables globales puede causar errores inesperados.
- Cualquier función puede cambiar el valor de una variable global $\Longrightarrow$efectos secundarios o laterales.
- Recomendaciones:
  - Evitar el uso de variables globales.
  - Mantener las variables lo más locales que se pueda.
  - Cuando se precise hacer accesible el valor de una variable a una función se pasará como argumento.

Variables automáticas

- Las variables locales que se definen en las funciones.
- Su ámbito es local.
- Su *vida* se restringe al tiempo en el que está activa la función.
- Los parámetros formales se tratan como variables automáticas.
- Se pueden especificar con la palabra reservada `auto` aunque no es necesario.

```
#include <stdio.h>

main()
{
   auto int valor; /* equivalente a int valor */

   valor = 5;
   printf("El valor es %d\n", valor);
}
```


Variables estáticas

- Su ámbito es local a la función.
- Su vida coincide con la del programa $\Longrightarrow$retienen sus valores durante toda la vida del programa.
- Se especifican con `static`.

```
#include <stdio.h>
void funcion(void);

main()
{
    funcion();
    funcion();
    funcion();
}

void funcion(void)
{
    static int veces = 0;

    veces = veces + 1;
    printf("Se ha llamado %d veces a funcion\n",
            veces);
}
```


Variables de tipo registro

- Informan al compilador que el programador desea que la variable se almacene en un lugar de rápido acceso, generalmente en registros.
- Si no existen registros disponibles se almacenará en memoria.
- Se especifican con `register`

```
#include <stdio.h>

main()
{
    register int j;

    for (j = 0; j < 10; j++)
        printf("Contador = %d\n", j);
}
```

Variables externas

- Variables globales.
- Hay que distinguir entre *definición* y *declaración* de variable externa.
- Una **definición** se escribe de la misma forma que las variables normales y reserva espacio para la misma en memoria.
- Una **declaración** no reserva espacio de almacenamiento $\Longrightarrow$se especifica con `extern`.
- Se emplean cuando un programa consta de varios módulos.
- En uno de ellos se define la variable.
- En los demás se declara (`extern`).

Ejemplo

- Modulo principal (`main.c`)

```
#include <stdio.h>

extern int valor;   /* se declara */
void funcion(void);
main()
{
   funcion();
   printf("Valor = %d\n", valor);
}
```

- Modulo auxiliar (`aux.c`)

```
int valor;      /* se define la variable */
void funcion(void)
{
   valor = 10;
}
```

- Se compila por separado:

```
gcc -c -Wall main.c
gcc -c -Wall aux.c
```

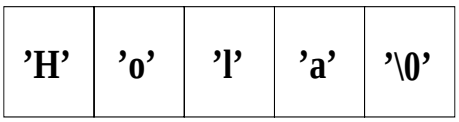
- Se obtienen dos módulos objetos: `main.o` y `aux.o`. El ejecutable (`prog`) se genera:

```
gcc main.o aux.o -o prog
```


CADENAS DE CARACTERES

Introducción

- Una **cadena de caracteres** es un conjunto o vector de caracteres.
- `á'` representa un caracter individual.
- `"Hola"` representa una cadena de caracteres.
- `"a"` representa una cadena de caracteres compuesta por un único caracter.
- Todas las cadenas de caracteres en C finalizan con el **caracter nulo** de C (`'\0'`).
- Este caracter indica el fin de una cadena de caracteres.
- La cadena `"hola"` se almacena en memoria:



- y su longitud es 4 (no se incluye el caracter nulo).

Declaración de cadenas de caracteres

```
char cadena[] = "Hola";
```

- Declara una cadena denominada `cadena` y reserva espacio para almacenar los siguientes caracteres:

  `'H'  ó'  'l'  á'  '\0'`

- Ejemplo:

```
#include <stdio.h>

main()
{
   char cadena[] = "hola";

   printf("La cadena es %s  \n", cadena);
   printf("Los caracteres son: \n");
   printf("%c \n", cadena[0]);
   printf("%c \n", cadena[1]);
   printf("%c \n", cadena[2]);
   printf("%c \n", cadena[3]);
   printf("%c \n", cadena[4]);
}
```

- `cadena[i]` representa el i-ésimo caracter de la cadena.

Declaración de cadenas de caracters (II)

- La declaración

```
char cadena[80];
```

- Declara una cadena de caracteres denominada `cadena` compuesta por 80 caracteres incluido el caracter nulo.
- La declaración

```
char cadena[4] = "Hola";
```

- Declara una cadena de exactamente 4 caracteres que no incluye el caracter nulo $\Longrightarrow$no se tratará de forma correcta como una cadena de caracteres.

Asignación de valores a cadenas de caracteres

- La asignación de valores iniciales a una cadena se puede realizar en su declaración:

```
char cadena[5] = "Hola";
char cadena[10] = "Hola";
```

- **NO** se puede asignar valores de la siguiente forma:

```
cadena = "Hola";
```

- Una forma de asignar un valor a una cadena es la siguiente:

```
strcpy(cadena, "Hola");
```

  – `cadena` debe tener suficiente espacio reservado.

- `strcpy(cadena1, cadena2);` copia `cadena1` en `cadena2` incluyendo el caracter nulo.
- `strcpy` se encuentra en el archivo de cabecera `string.h`

Lectura y escritura de cadenas de caracteres

```
#include <stdio.h>
#define TAM_CADENA    80

main()
{
   char cadena[TAM_CADENA];

   printf("Introduzca una cadena: ");

   scanf("%s", cadena);

   printf("La cadena es %s\n", cadena);
}
```

- `scanf` deja de buscar cuando encuentra un blanco $\Longrightarrow$si se introduce

  `Hola a todos`

- solo se leerá `Hola`.
- No es necesario el operador de dirección (`&`) ya que `cadena` representa de forma automática la dirección de comienzo.

Lectura y escritura de cadenas de caracteres (II)

- La función `gets` lee una línea completa hasta que encuentre el retorno de carro incluyendo los blancos.
- La función `puts` escribe una cadena de caracteres junto con un salto de línea.

```
#include <stdio.h>
#define TAM_LINEA     80

main()
{
   char linea[TAM_LINEA];

   printf("Introduzca una linea: \n");
   gets(linea);

   puts("La linea es");
   puts(linea);
}
```

```
puts("La linea es:");
```

- es equivalente a:

```
printf("La linea es: \n");
```


Ejemplo

- Programa que lee líneas hasta fin de fichero y cuenta el número de caracteres de cada línea:

```
#include <stdio.h>
#define TAM_LINEA   80

main()
{
   char linea[TAM_LINEA];
   int num_car;
   int j;

   while (gets(linea) != NULL)
   {
      num_car = 0;
      j = 0;
      while (linea[j] != '\0')
      {
         num_car ++;
         j++;
      }

      printf("Esta linea tiene %d caracteres\n",
              num_car);
   }
}
```


Paso de cadenas de caracteres a funciones

- Cuando se pasa una cadena a una función se pasa la dirección de comienzo de la misma $\Longrightarrow$la cadena se puede modificar en la función.
- Ejemplo:

```
#include <stdio.h>
#define TAM_LINEA    80
void leer_linea(char linea[]);

main()
{
   char linea[TAM_LINEA];

   leer_linea(linea);
   puts("La linea es");
   puts(linea);
}

void leer_linea(char linea[])
{
   gets(linea);
   return;
}
```


Ejemplo

- Programa que lee líneas hasta fin de fichero y cuenta el número de caracteres de cada línea:

```
#include <stdio.h>
#define TAM_LINEA   80
int longitud(char cadena[]);
main()
{
   char linea[TAM_LINEA];
   int num_car;

   while (gets(linea) != NULL)
   {
      num_car = longitud(linea);
      printf("Esta linea tiene %d caracteres\n",
              num_car);
   }
}
int longitud(char cadena[])
{
   int j = 0;

   while (cadena[j] != '\0')
      j++;
   return(j);
}
```

Ejemplo

- Programa que lee una línea en minúsculas y la convierte a mayúsculas.

```
#include <stdio.h>
#include <ctype.h>
#define TAM_LINEA   80
void Convertir_may(char min[], char may[]);

main()
{
   char linea_min[TAM_LINEA];
   char linea_may[TAM_LINEA];

   while (gets(linea_min) != NULL)
   {
      Convertir_may(linea_min, linea_may);
      puts(linea_may);
   }
}
```



```
void Convertir_may(char min[], char may[])
{
   int j = 0;

   while (min[j] != '\0')
   {
      may[j] = toupper(min[j]);
      j++;
   }
  may[j] = '\0';
  return;
}
```


Funciones de biblioteca para manejar cadenas

- En `<string.h>`

Función	Significado
`strcpy`	Copia una cadena en otra
`strlen`	Longitud de la cadena
`strcat`	Concatenación de cadenas
`strcmp`	Comparación de dos cadenas
`strchr`	Buscar un caracter dentro de una cadena
`strstr`	Buscar una cadena dentro de otra

- En `<stdlib.h>`

Función	Significado
`atoi`	Convierte una cadena a un entero (`int`)
`atol`	Convierte una cadena a un entero largo (`long`)
`atof`	Convierte una cadena a un real (`double`)

VECTORES Y MATRICES

Introducción

- Un vector o *array* es un conjunto de valores, todos del mismo tipo, a los que se da un nombre común, distinguiendo cada uno de ellos por su índice.
- Coincide con el concepto matemático de vector:

$$V = (V_0, V_1, \ldots V_n)$$

- El número de índices determina la dimensión del vector.
- En C los datos individuales de un vector pueden ser de cualquier tipo (`int`, `char`, `float`, etc.)

Definición de un vector

- Un vector unidimensional se declara:

```
tipo_dato vector[expresion];
```

  donde

  - `tipo_dato` es el tipo de datos de cada elemento.
  - `vector` es el nombre del vector (*array*).
  - `expresion` indica el número de elementos del vector.

- Ejemplos:

```
int v_numeros[20];
float n[12];
char vector_letras[5];
```

- En C el primer índice del vector es **0**.
- Sólo se puede asignar valores iniciales a vectores estáticos y globales.

```
int n[5] = {1, 2, 18, 24, 3};
```

- ISO C también permite asignar valores iniciales a un vector usando expresiones constantes.

Procesamiento de un vector

- En C no se permiten operaciones que impliquen vectores completos $\Longrightarrow$
  - No se pueden asignar vectores completos.
  - No se pueden comparar vectores completos.
- El procesamiento debe realizarse elemento a elemento.

```
#include <stdio.h>
#define TAM_VECTOR   10

main()
{
   int vector_a[TAM_VECTOR];
   int vector_b[TAM_VECTOR];
   int j; /* variable utilizada como indice */

   /* leer el vector a */
   for (j = 0; j < TAM_VECTOR; j++)
   {
      printf("Elemento %d: ", j);
      scanf("%d", &vector_a[j]);
   }
```



```
   /* copiar el vector */
   for (j = 0; j < TAM_VECTOR; j++)
      vector_b[j] = vector_a[j];

   /* escribir el vector b */
   for (j = 0; j < TAM_VECTOR; j++)
      printf("El elemento %d es %d \n",
              j, vector_b[j]);
}
```

Paso de vectores a funciones

- Un vector se pasa a una función especificando su nombre sin corchetes.
- El nombre representa la dirección del primer elemento del vector $\Longrightarrow$los vectores se pasan por **referencia** y se pueden modificar en las funciones.
- El argumento formal correspondiente al vector se escribe con un par de corchetes cuadrados vacíos. El tamaño no se especifica.

Ejemplo

- Programa que calcula la media de los componentes de un vector.

```
#include <stdio.h>
#define MAX_TAM 4

void leer_vector(int vector[]);
int media_vector(int vector[]);

main()
{
   int v_numeros[MAX_TAM];
   int media;

   leer_vector(v_numeros);
   media = media_vector(v_numeros);

   printf("La media es %d\n", media);
}
```



```
void leer_vector(int vector[])
{
   int j;

   for(j=0; j<MAX_TAM; j++)
   {
      printf("Elemento %d: ", j);
      scanf("%d", &vector[j]);
   }
   return;
}


int media_vector(int vector[])
{
   int j;
   int media = 0;

   for(j=0; j<MAX_TAM; j++)
      media = media + vector[j];

   return(media/MAX_TAM);
}
```


Punteros y vectores

- El nombre del vector representa la dirección del primer elemento del vector

```
float vector[MAX_TAM];

vector == &vector[0]
```

- El nombre del vector es realmente un puntero al primer elemento del vector.

  `&x[0]` $\Longleftrightarrow$`x`

  `&x[1]` $\Longleftrightarrow$`(x+1)`

  `&x[2]` $\Longleftrightarrow$`(x+2)`

  `&x[i]` $\Longleftrightarrow$`(x+i)`

- Es decir, `&x[i]` y `(x+i)` representan la dirección del i-ésimo elemento del vector `x` $\Longrightarrow$
- `x[i]` y `*(x+1)` representan el contenido del i-ésimo elemento del vector `x`.

- Cuando un vector se define como un puntero no se le pueden asignar valores ya que un puntero no reserva espacio en memoria.
- `float x[10]` define un vector compuesto por 10 números reales $\Longrightarrow$reserva espacio para los elementos.
- `float *x` declara un puntero a float. Si se quiere que `float x` se comporte como un vector $\Longrightarrow$habrá que reservar memoria para los 10 elementos:

```
x = (float *) malloc(10 * sizeof(float));
```

- `malloc(nb)` (`stdlib.h`) reserva un bloque de memoria de `nb` bytes.
- Para liberar la memoria asignada se utiliza `free()` (`stdlib.h`)

```
free(x);
```

- El uso de punteros permite definir vectores de forma dinámica.

Ejemplo

- Programa que calcula la media de un vector de tamaño especificado de forma dinámica.

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

void leer_vector(int vector[], int dim);
int media_vector(int vector[], int dim);

main()
{
   int *v_numeros;
   int dimension;
   int media;

   printf("Dimension del vector: ");
   scanf("%d", &dimension);

   v_numeros = (int *) malloc(dimension*sizeof(int));

   leer_vector(v_numeros, dimension);
   media = media_vector(v_numeros, dimension);

   printf("La media es %d\n", media);
   free(v_numeros);
}
```



```
void leer_vector(int vector[], int dim)
{
   int j;

   for(j=0; j<dim; j++)
   {
      printf("Elemento %d: ", j);
      scanf("%d", &vector[j]);
   }
   return;
}


int media_vector(int vector[], int dim)
{
   int j;
   int media = 0;

   for(j=0; j<dim; j++)
      media = media + vector[j];

   return(media/dim);
}
```


- El programa anterior es equivalente al siguiente:

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

int *crear_vector(int dim);
void leer_vector(int *vector, int dim);
int media_vector(int *vector, int dim);

main()
{
   int *v_numeros;
   int dimension;
   int media;

   printf("Dimension del vector: ");
   scanf("%d", &dimension);

   v_numeros = crear_vector(dimension);
   leer_vector(v_numeros, dimension);
   media = media_vector(v_numeros, dimension);

   printf("La media es %d\n", media);
   free(v_numeros);
}
```

```
int *crear_vector(int dim)
{
   int *vec;

   vec = (int *) malloc(dim*sizeof(int));
   return(vec);
}
void leer_vector(int *vector, int dim)
{
   int j;

   for(j=0; j<dim; j++)
   {
      printf("Elemento %d: ", j);
      scanf("%d", (vector + j));
   }
   return;
}
int media_vector(int *vector, int dim)
{
   int j;
   int media = 0;

   for(j=0; j<dim; j++)
      media = media + *(vector + j);

   return(media/dim);
}
```


Vectores y cadenas de caracteres

- Una cadena de caracteres es un vector de caracteres $\Longrightarrow$cada elemento del vector almacena un caracter.
- **Ejemplo:** Función que copia una cadena en otra:

```
void copiar(char *destino, char *fuente)
{
   while (*fuente != '\0')
   {
      *destino = *fuente;
      destino ++;
      fuente++ ;
   }
   *destino = '\0';
   return;
}
```


Vectores multidimensionales

- Un vector multidimensional se declara:

  ```
  tipo_dato vector[exp1] [exp2] ... [expN];
  ```

- Matrices o vectores de 2 dimensiones:

  `int matriz[20][30];`

  define una matriz de 20 filas por 30 columnas.

- El elemento de la fila `i` columna `j` es `matriz[i][j]`

Ejemplo

- Función que calcula el producto de dos matrices cuadradas.

```
void multiplicar(float a[][DIMENSION],
                 float b[][DIMENSION],
                 float c[][DIMENSION])
{
   int i, j, k;

   for(i = 0; i < DIMENSION; i++)
      for(j = 0; j < DIMENSION; j++)
      {
         c[i][j] = 0.0;
         for(k = 0; k < DIMENSION; k++)
            c[i][j] += a[i][k] * b[k][j];
      }
   return;
}
```

Punteros y vectores multidimensionales

- Programa que define de forma dinámica una matriz (mediante punteros).

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

float **crear_matriz(int fil, int col);
void destruir_matriz(float **mat, int fil);
void leer_matriz(float **mat, int fil, int col);
void imprimir_matriz(float **mat, int fil, int col);

main()
{
   float **matriz;
   int fil, col;

   printf("Numero de filas: ");
   scanf("%d", &fil);

   printf("Numero de columnas: ");
   scanf("%d", &col);

   matriz = crear_matriz(fil, col);
   leer_matriz(matriz, fil, col);
   imprimir_matriz(matriz, fil, col);
   destruir_matriz(matriz, fil);
}
```



```
float **crear_matriz(int fil, int col)
{
   int j;
   float **mat;

   mat = (float **) malloc(fil * sizeof(float *));
   for (j = 0; j < fil; j++)
      mat[j] = (float *) malloc(col * sizeof(float));

   return(mat);
}

void destruir_matriz(float **mat, int fil)
{
   int j;

   for (j = 0; j < fil; j++)
      free(mat[j]);
   free(mat);
   return;
}
```



```
void leer_matriz(float **mat, int fil, int col)
{
   int i, j;
   float dato;

   for (i = 0; i < fil; i++)
      for (j = 0; j < col; j++)
      {
         printf("Elemento %d %d: ", i,j);
         scanf("%f", &dato);
         *(*(mat + i) + j) = dato;
      }
   return;
}

void imprimir_matriz(float **mat, int fil, int col)
{
   int i, j;
   for (i = 0; i < fil; i++)
      for (j = 0; j < col; j++)
         printf("Elemento %d %d = %f\n",
                 i,j,mat[i][j]);
   return;
}
```


ESTRUCTURAS DE DATOS

Estructuras

- Es una estructura de datos compuesta de elementos individuales que pueden ser de distinto tipo.
- Cada uno de los elementos de una estructura se denomina **miembro**.
- Definición de una estructura:

```
struct nombre_estructura
{
   tipo1 miembro_1;
   tipo2 miembro_2;
   .
   .
   tipoN miembro_N;
};
```

- Los miembros pueden ser de cualquier tipo excepto `void`.

Ejemplos

```
struct fecha
{
   int mes;
   int dia;
   int anno;
};
```

- declara una estructura denominada fecha.

```
struct fecha fecha_de_hoy;
```

- declara una variable denominada `fecha_de_hoy` de tipo `struct fecha`.

```
struct cuenta
{
   int cuenta_no;
   int cuenta_tipo;
   char nombre[80];
   float saldo;
   struct fecha ultimopago;
};
```


Procesamiento de una estructura

- Los miembros de una estructura se procesan individualmente.
- Para hacer referencia a un miembro determinado:

```
variable_estructura.miembro
```

- El `.` se denomina **operador de miembro**.
- Ejemplo: imprimir la fecha de hoy

```
struct fecha hoy;

printf("%d: %d: %d\n", hoy.dia,
        hoy.mes, hoy.anno);
```

- Se pueden copiar estructuras:

```
struct fecha hoy, ayer;

ayer = hoy;
```

- No se pueden comparar estructuras.

Ejemplo

```
#include <stdio.h>

struct fecha
{
   int dia;
   int mes;
   int anno;
};

struct cuenta
{
   int cuenta_no;
   char nombre[80];
   float saldo;
   struct fecha ultimo_pago;
};
```

```
main()
{
   struct cuenta c1, c2;

   /* rellena la estructura c1 */
   c1.cuenta_no = 2;
   strcpy(c1.nombre, "Pepe");
   c1.saldo = 100000;
   c1.ultimo_pago.dia = 12;
   c1.ultimo_pago.mes = 5;
   c1.ultimo_pago.anno = 1997;

   /* asignacion de estructuras */
   c2 = c1;

   printf("No. Cuenta %d \n", c2.cuenta_no);
   printf("Nombre %s \n", c2.nombre);
   printf("Saldo %f \n", c2.saldo);
   printf("Fecha de ultimo pago: %d:%d:%d \n",
            c2.ultimo_pago.dia, c2.ultimo_pago.mes,
            c2.ultimo_pago.anno);
}
```


Paso de estructuras a funciones

- Se pueden pasar miembros individuales y estructuras completas.
- Las estructuras se pasan por valor.
- Una función puede devolver una estructura.
- **Ejemplo**: función que imprime una fecha.

```
void imprimir_fecha(struct fecha f)
{
   printf("Dia: %d\n", f.dia);
   printf("Mes: %d\n", f.mes);
   printf("Anno: %d\n", f.anno);
   return;
}
```

- **Ejemplo**: función que lee una fecha

```
struct fecha leer_fecha(void)
{
   struct fecha f;

   printf("Dia: ");
   scanf("%d", &(f.dia));

   printf("Mes: ");
   scanf("%d", &(f.mes));

   printf("Anno: ");
```



```
      scanf("%d", &(f.anno));
      return(f);
   }
```

- En el programa principal:

```
main()
{
   struct fecha fecha_de_hoy;

   fecha_de_hoy = leer_fecha();
   imprimir_fecha(fecha_de_hoy);
}
```


Punteros a estructuras

Igual que con el resto de variables.

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

main()
{
    struct punto punto_1;
    struct punto *punto_2;

    punto_1.x = 2.0;
    punto_1.y = 4.0;

    punto_2 = &punto_1;

    printf("x = %f \n", punto_2->x);
    printf("y = %f \n", punto_2->y);
}
```

- En una variable de tipo puntero a estructura los miembros se acceden con `->`

Punteros a estructuras (II)

- Se pueden pasar punteros a funciones $\Longrightarrow$se pasan por referencia.
- Un puntero a una estructura no reserva memoria.

```
void leer_punto(struct punto *p);
void imprimir_punto(struct punto p);

main()
{
   struct punto *p1;

   p1 = (struct punto *)malloc(sizeof(struct punto));

   leer_punto(p1);
   imprimir_punto(*p1);
   free(p1);
}
```



```
void leer_punto(struct punto *p)
{
   printf("x = ");
   scanf("%f", &(p->x));
   printf("y = ");
   scanf("%f", &(p->y));
}

void imprimir_punto(struct punto p)
{
   printf("x = %f\n", p.x);
   printf("y = %f\n", p.y);
}
```


Vectores de estructuras

- Se pueden definir vectores cuyos elementos sean estructuras:

```
main()
{
   struct punto vector_puntos[10];
   int j;

   for(j = 0; j < 10; j++)
   {
      printf("x_%d = %f\n", j, vector_puntos[j].x);
      printf("y_%d = %f\n", j, vector_puntos[j].y);
   }
}
```


Uniones

- Una `unión` contiene miembros cuyos tipos de datos pueden ser diferentes (igual que las estructuras).
- Su declaración es similar a las estructuras:

```
union nombre_estructura
{
   tipo1 miembro_1;
   tipo2 miembro_2;
   .
   .
   tipoN miembro_N;
};
```

- Todos los miembros que componene la unión comparten la misma zona de memoria
- Una variable de tipo `unión` sólo almacena el valor de uno de sus miembros.

Ejemplo

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

union numero
{
   int entero;
   float real;
};

main()
{
   union numero num;

   /* leer un entero e imprimirlo */
   printf("Entero: ");
   scanf("%d", &(num.entero));
   printf("El entero es %d\n", num.entero);

   /* leer un real e imprimirlo */
   printf("Real: ");
   scanf("%f", &(num.real));
   printf("El entero es %f\n", num.real);
}
```


Tipos enumerados

- Un tipo enumerado es similar a las estructuras.
- Sus miembros son constantes de tipo `int`.
- Definición:

```
enum nombre {m1, m2, ..., mN};
```

- Ejemplo:

```
enum color {negro, blanco, rojo};
```


Ejemplo

```
#include <stdio.h>
enum diasemana {lunes,martes,miercoles,jueves,
           viernes,sabado,domingo};
main()
{
   enum diasemana dia;

   for(dia=lunes; dia<=domingo; dia++)
      switch(dia)
      {
         case lunes:
                printf("Lunes es laboral\n");
                break;
         case martes:
                printf("Martes es laboral\n");
                break;
         case miercoles:
                printf("Miercoles es laboral\n");
                break;
         case jueves:
                printf("Jueves es laboral\n");
                break;
         case viernes:
                printf("Viernes es laboral\n");
                break;
         case sabado:
                printf("Sabado no es laboral\n");
                break;
```



```
         case domingo:
                printf("Domingo no es laboral\n");
                break;
      }
}
```

Definición de tipos de datos (typedef)

- Un nuevo tipo se define como:

```
typedef tipo nuevo_tipo;
```

- Ejemplos:

```
typedef char letra;
typedef struct punto tipo_punto;


letra c;
tipo_punto p;
```


Estructuras de datos autorreferenciadas

- Estructura que contiene un puntero a una estructura del mismo tipo.

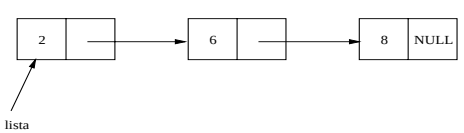
```
struct nombre_estructura
{
   tipo1 miembro_1;
   tipo2 miembro_2;
   .
   .
   tipoN miembro_N;
   struct nombre_estructura *puntero;
};
```

- Ejemplo: listas enlazadas.

Listas enlazadas

- Es una estructura de datos compuesta por un conjunto de elementos enlazados.
- Lista de numero enteros:

```
struct elemento
{
   int num;
   struct elemento *enlace;
};

typedef struct elemento lista_num;

lista_num *lista;
```



- Una lista es una estructura de datos dinámica que puede crecer según las necesidades del programa (a diferencia de los vectores).

Ejemplo de lista enlazada

```
#include <stdio.h>
#include <stdlib.h>
struct elemento
{
   int num;
   struct elemento *enlace;
};
typedef struct elemento lista_num;

void insertar(lista_num **ptr, int num);
void mostrar(lista_num *ptr);
void borrar(lista_num *ptr);
main()
{
   lista_num *lista = NULL;
   int numero;

   scanf("%d", &numero);
   while(numero != 0)
   {
      insertar(&lista, numero);
      scanf("%d", &numero);
   }
   mostrar(lista);
   borrar(lista);
}
```

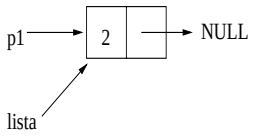
```
void insertar(lista_num **ptr, int num)
{
   lista_num *p1, *p2;

   p1 = *ptr;
   if (p1 == NULL)
   {
      p1 = (lista_num *)malloc(sizeof(lista_num));
      p1->num = num;
      p1->enlace = NULL;
      *ptr = p1;
   }
   else
   {
      while (p1->enlace != NULL)
         p1 = p1->enlace;

      p2 = (lista_num *)malloc(sizeof(lista_num));
      p2->num = num;
      p2->enlace = NULL;
      p1->enlace = p2;
   }
}
```



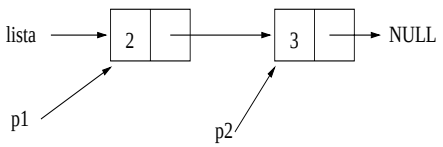
```
void mostrar(lista_num *ptr)
{
   while (ptr != NULL)
   {
      printf("%d \n", ptr->num);
      ptr = ptr->enlace;
   }
}

void borrar(lista_num *ptr)
{
   lista_numeros *aux;

   while (ptr != NULL)
   {
      aux = ptr->enlace;
      free(ptr);
      ptr = aux;
   }
}
```


1. Inicialmente:

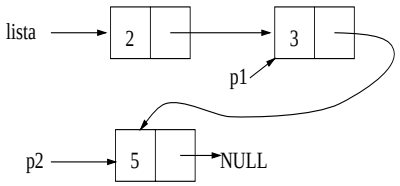
2. insertar(&lista, 2);



3. insertar(&lista, 3);



4. Insertar(&lista, 5);



Diferencia entre vectores y listas

- Un vector definido de la siguiente forma:

  ```
  int vector[10];
  ```

  reserva un número fijo de elementos antes de la compilación.

- Un vector creado de la siguiente forma:

  ```
  int *vector;
  vecto = (int *)malloc(DIM*sizeof(int));
  ```

  reserva un número fijo de elementos que puede variar de una ejecución a otra (en función de DIM).

- Una lista de enteros puede almacenar un número que puede variar durante la ejecución (no es fijo).

ENTRADA/SALIDA

Apertura y cierre de un archivo

- Para abrir un archivo:

```
desc = fopen(nombre_archivo, modo);
```

donde `desc` se declara como:

```
FILE *desc;
```

y `modo` especifica la forma de apertura del archivo.
si `fopen` devuelve `NULL` el fichero no se pudo abrir.

Modo	Significado
`"r"`	Abre un archivo existente para lectura.
`"w"`	Abre un nuevo archivo para escritura. Si existe el archivo se borra su contenido. Si no existe se crea.
`"a"`	Abre un archivo existente para añadir datos al final. Si no existe se crea.
`"r+"`	Abre un archivo existente para lectura y escritura.
`"w+"`	Abre un archivo nuevo para escritura y lectura. Si existe lo borra. Si no existe lo crea.
`"a+"`	Abre un archivo para leer y añadir.

- Para cerrar el fichero:

```
fclose(desc);
```


Ejemplo

```
#include <stdio.h>

main()
{
   FILE *desc;

   desc = fopen("ejemplo.txt", "w");
   if (desc == NULL)
   {
      printf("Error, no se puede abrir el archivo\n");
   }
   else
   {
      /* se procesa el archivo */

      /* al final se cierra */
      fclose(desc);
   }
   exit(0);
}
```

Lectura y escritura

- Para leer:

```
fscanf(desc, formato, ...);
```

- Para escribir:

```
fprintf(desc, formato, ...);
```

- Ejemplo:

```
fscanf(desc, "%d %f", &num1, &num2);

fprintf(desc, "%d\n", num1);
```


Ejemplo

- Programa que copia un archivo en otro.

```
#include <stdio.h>

main()
{
   FILE *fent;
   FILE *fsal;
   char car;

   fent = fopen("entrada.txt", "r");
   if (fent == NULL)
   {
      printf("Error abriendo entrada.txt \n");
      exit(0);
   }

   fsal = fopen("salida.txt", "w");
   if (fsal == NULL)
   {
      printf("Error creando entrada.txt \n");
      close(fent);
      exit(0);
   }
```



```
   while (fscanf(fent, "%c", &car) != EOF)
      fprintf(fsal, "%c", car);

   fclose(fent);
   fclose(fsal);
   exit(0);
}
```


Argumentos en la línea de mandatos

- Cuando se ejecuta un programa se pueden pasar argumentos a la función `main` desde la línea de mandatos.

```
nombre_programa arg1 arg2 arg3 ... argn
```

- El prototipo de la función `main` es el siguiente:

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

- `argc` indica el número de argumentos que se pasa incluido el el nombre del programa.
- `argv[0]` almacena el nombre del programa.
- `argv[1]` almacena el primer argumento que se pasa.
- `argv[i]` almacena el i-ésimo argumento que se pasa.

Ejemplo

- Programa que recibe como argumento el nombre de dos archivos compuestos de números enteros:

  `copiar_enteros archivo_entrada archivo_salida`

  el programa copia `archivo_entrada` en `archivo_salida`.


```
#include <stdio.h>

main(int argc, char *argv[])
{
   FILE *fent;
   FILE *fsal;
   int num;

   if (argc != 3)
   {
      printf(Üso: %s fich_entrada fich_salida\n",
                  argv[0]);
      exit(0);
   }

   fent = fopen(argv[1], "r");
   if (fent == NULL)
   {
      printf("Error abriendo entrada.txt \n");
      exit(0);
   }
```



```
   fsal = fopen(argv[2], "w");
   if (fsal == NULL)
   {
      printf("Error creando entrada.txt \n");
      close(fent);
      exit(0);
   }

   while (fscanf(fent, "%d", &num) != EOF)
      fprintf(fsal, "%d\n", num);

   fclose(fent);
   fclose(fsal);
   exit(0);
}
```


ASPECTOS AVANZADOS

Operadores de bits

- Permiten manejar los bits individuales en una palabra de memoria.
- Categorías:
  - Operador de complemento a uno.
  - Operadores lógicos binarios.
  - Operadores de desplazamiento.

Operador de complemento a uno

- Operador de complemento a uno $\Longrightarrow$(`~`).
- Invierte los bits de su operando $\Longrightarrow$los unos se transforman en ceros y los ceros en unos.
- Ejemplo:

```
#include <stdio.h>

main()
{
    unsigned int n = 0x4325;

    printf("%x ; %x\n", n, ~n);
}
```

- En una máquina de 32 bits:
  - `n = 0x4325`
  - `~n = 0xffffbcda`

Operadores lógicos binarios

- Operadores lógicos binarios de C:

Operador	Función
`&`	AND binario
`|`	OR binario
`^`	OR exclusivo binario

- Las operaciones se realizan de forma independiente en cada par de bits que corresponden a cada operando.

a	b	a & b	a | b	a ^ b
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

- Primero se comparan los bits menos significativos, luego los siguientes bits menos significativos y así sucesivamente.

Ejemplo

```
#include <stdio.h>
main()
{
    int a = 8;
    int b = 5;
    int c = 3;


    printf("%d\n", a & b);
    printf("%d\n", b & c);
    printf("%d\n", a | c);
    printf("%d\n", b ^ c);


}
```

- Este programa imprime: 0, 1, 11, 6

Máscaras

- El *enmascaramiento* es un proceso en el que un patrón dado de bits se convierte en otro patrón por medio de una operación lógica a nivel de bits.
- El segundo operando se llama **máscara**.
- Ejemplo: programa que obtiene los cuatros bits menos significativos de un número dado.

```
#include <stdio.h>

#define MASCARA 0xF

main()
{
   int n;

   printf("Introduzca n: ");
   scanf("%d", &n);
   printf("bits menos signif. = %d\n",
               n & MASCARA );
}
```


Operadores de desplazamiento

- Desplazamiento a la izquierda: `<<`
- Desplazamiento a la derecha: `>>`
- Requieren dos operandos.
- El primero es un operando de tipo entero que representa el patrón de bits a desplazar.
- El segundo es un entero sin signo que indica el número de desplazamientos .
- **Ejemplo.** Si $a = 8 \Longrightarrow$

Expresión	**Valor**
$a << 1$	16
$a << 2$	32
$a << 3$	64
$a >> 3$	1

- **Operadores de asignación binarios**

  `&=` `^=` `|=` `<<=` `>>=`

Campos de bits

- C permite descomponer una palabra en distintos campos de bits que pueden manipularse de forma independiente.
- La descomposición puede realizarse:

```
struct marca
{
   miembro_1 : i;
   miembro_2 : j;
   .
   .
   miembro_N : k;
};
```

- Cada declaración de miembro puede incluir una especificación indicando el tamaño del campo de bits correspondiente (i, j, k).
- ISO C permite que los campos de bits sean de tipo *unsigned int*, *signed int* o *int*.
- La ordenación de los campos de bits puede variar de un compilador a otro.

Ejemplo

```
#include <stdio.h>

struct registro
{
   unsigned a : 2;
   unsigned b : 4;
   unsigned c : 1;
   unsigned d : 1;
};

main()
{
  struct registro r;

  r.a = 5; r.b = 2; r.c = 1; r.d = 0;
  printf("%d %d %d %d\n", r.a, r.b, r.c, r.c);
  printf("r requiere %d\n", sizeof(r));
}
```

Punteros a funciones

- C permite definir punteros a funciones.
- La forma de declarar un puntero a una función es:

```
tipo_dato (*nombre_funcion)(argumentos);
```

- La forma de llamar a la función a partir de su puntero es:

```
(*nombre_funcion)(argumentos);
```


Ejemplo

```
#include <stdio.h>
#define FALSE    0
#define TRUE     1

int sumar(int a, int b)
{
    return(a+b);
}
int restar(int a, int b)
{
    return(a-b);
}

main()
{
   int (*operacion)(int a, int b);
   int a, b, oper, error, res;

   printf("a, b: ");
   scanf("%d %d", &a, &b);
   printf("Operacion (0=suma, 1=resta): ");
   scanf("%d", &oper);
```



```
   error = FALSE;
   switch(oper)
   {
      case 0: operacion = sumar;
              break;
      case 1: operacion = restar;
              break;
      default: error = TRUE;
   }
   if (!error)
   {
      res = (*operacion)(a,b);
      printf("Resultado = %d\n", res);
   }
}
```


Ejemplo

- Programa que ejecuta la secuencia F0 0F C7 C8.
- Si se ejecuta en un Pentium bloquea la máquina.

```
#include <stdio.h>

char x [5] = { 0xf0, 0x0f, 0xc7, 0xc8 };

main ()
{
    void (*f)() = x;
    f();
}
```

Funciones como argumentos

- En C se pueden pasar funciones como argumentos a otras funciones.
- Cuando una función acepta el nombre de otra como argumento, la declaración formal debe identificar este argumento como un puntero a otra función.
- Ejemplo:

```
#define FALSE    0
#define TRUE     1
int sumar(int a, int b)
{
    return(a+b);
}

int restar(int a, int b)
{
    return(a-b);
}
```


Ejemplo II

```
int ejecutar(int (*f)(int a, int b),
                int a, int b)
{
   int res;
   res = (*f)(a,b);
   return(res);
}
main()
{
   int a, b, oper, error, res;

   printf("a, b: ");
   scanf("%d %d", &a, &b);
   printf("Operacion (0=suma, 1=resta): ");
   scanf("%d", &oper);
   error = FALSE;
   switch(oper)
   {
      case 0: res = ejecutar(sumar, a, b); break;
      case 1: res = ejecutar(restar, a, b); break;
      default: error = TRUE;
   }
   if (!error)
      printf("Resultado = %d\n", res);
}
```


Funciones con Nº variable de argumentos

- C permite la declaración de funciones con un número de argumentos variable.
- Ejemplo: `printf()`, `scanf()`.
- El prototipo de estas funciones es:

```
int sumar(int contador, ...);
```

- El primer argumento es fijo y obligatorio.
- Hay que utilizar las funciones y macros definidas en `<stdarg.h>`
- `va_list`, variable local utilizada para acceder a los parámetros.
- `va_start`, inicializa los argumentos.
- `va_arg`, obtiene el valor del siguiente parámetro.
- `va_end`, debería ser llamada una vez procesados todos los argumentos.

Ejemplo

- Función que suma un número variable de enteros.

```
int sumar(int contador, ...)
{
   va_list ap;
   int arg;
   int total = 0;
   int i = 0;

   va_start(ap, contador);
   for (i = 0; i < contador; i++)
   {
      arg = va_arg(ap, int);
      total = total + arg;
   }
   va_end(ap);
   return(total);
}
```

Ejemplo

- Función que imprime un conjunto indeterminado de cadenas de caracteres:

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

void imprimir(char  *s1, ...)
{
   va_list ap;
   char *arg;

   va_start(ap, s1);
   printf("%s\n", s1);
   while ((arg = va_arg(ap, char *)) != NULL)
      printf("%s\n", arg);
   va_end(ap);
   return;
}
main()
{

   imprimir("1", "2", "3", NULL);
}
```


Ejemplo

Función para imprimir argumentos.

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

enum tipos_arg {INTARG, FLOATARG, NULO};
void miprint(enum tipos_arg *args, ...)
{
  va_list ap;
  int tipo;

  va_start(ap, args);
  while((tipo = *args++) != NULO)
  {
     switch(tipo)
     {
         case INTARG:
           printf("int %d\n", va_arg(ap,int));
           break;
         case FLOATARG:
           printf("float %f\n", va_arg(ap,double));
           break;
         default:
           printf("Tipo desconocido\n");
     }
  }
  va_end(ap);
  return;
}
```



```
main()
{
  enum tipos_arg args[6] = {INTARG,
                            INTARG,
                            FLOATARG,
                            INTARG,
                            NULO};

  miprint(&args[0], 5, 2, 3.4, 6);
}
```


Compilación condicional

- Las siguientes líneas:

```
#ifdef     identificador
#ifndef    identificador
```

  indican la compilación condicional del texto que las sigue hasta encontrar la línea:

```
#endif
```

- Para compilar con este tipo de directivas:

```
cc -Didentificador programa.c
```

- La compilación condicional es útil en la fase de depuración y prueba de un programa.

Ejemplo

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

int sumar(int contador, ...)
{
   va_list ap;
   int arg;
   int total = 0;
   int i = 0;

   va_start(ap, contador);
   for (i = 0; i < contador; i++)
   {
      arg = va_arg(ap, int);
#ifdef DEBUG
      printf("total = %d\n", total);
      printf("arg = %d\n", arg);
#endif
      total = total + arg;
   }
   va_end(ap);
   return(total);
}
```



```
main()
{
   int suma;

   suma = sumar(4, 1, 2, 3, 4);
   printf("suma = %d\n", suma);
}
```


HERRAMIENTAS PARA EL DESARROLLO DE PROGRAMAS EN UNIX

ar: Gestor de bibliotecas

- Utilidad para la creación y mantenimiento de bibliotecas de ficheros.
- Su principal uso es crear bibliotecas de objetos: Agrupar un conjunto de objetos relacionados.
- En la línea de compilación se especifica la biblioteca (por convención `libnombre.a`) en vez de los objetos.
- El enlazador extraerá de la biblioteca los objetos que contienen las variables y funciones requeridas.
- Formato del mandato:

  ```
  ar opciones biblioteca ficheros...
  ```

- Algunas opciones:
  - -d Elimina de la biblioteca los ficheros especificados
  - -r Añade (o reemplaza si existe) a la biblioteca los ficheros especificados. Si no existe la biblioteca se crea
  - -ru Igual que -r pero sólo reemplaza si el fichero es más nuevo
  - -t Muestra una lista de los ficheros contenidos en la biblioteca
  - -v *Verbose*
  - -x Extrae de la biblioteca los ficheros especificados

- Ejemplos:
  - Obtención de la lista de objetos contenidos en la biblioteca estándar de C

    ```
    ar -tv /usr/lib/libc.a
    ```

  - Creación de una biblioteca con objetos que manejan distintas estructuras de datos

    ```
    ar -rv $HOME/lib/libest.a pila.o lista.o
    ar -rv $HOME/lib/libest.a arbol.o hash.o
    ```

  - Dos formas de compilar un programa que usa la biblioteca matemática y la creada

    ```
    cc -o pr pr.c -lm $HOME/lib/libest.a
    cc -o pr pr.c -L$HOME/lib -lm -lest
    ```


gdb: Depurador de programas

- Permite que el usuario pueda controlar la ejecución de un programa y observar su comportamiento interno mientras ejecuta.
- El programa debe compilarse con la opción -g.
- Algunas de sus funciones:
  - Establecer puntos de parada en la ejecución del programa (*breakpoints*)
  - Examinar el valor de variables
  - Ejecutar el programa línea a línea
- Formato del mandato:

  ```
  gdb programa
  ```

- Algunos mandatos del gdb
  - `run`: Arranca la ejecución del programa
  - `break`: Establece un *breakpoint* (un número de línea o el nombre de una función)
  - `list`: Imprime las líneas de código especificadas
  - `print`: Imprime el valor de una variable
  - `continue`: Continúa la ejecución del programa después de un *breakpoint*
  - `next`: Ejecuta la siguiente línea. Si se trata de una llamada a función, la ejecuta completa

  - `step`: Ejecuta la siguiente línea. Si se trata de una llamada a función, ejecuta sólo la llamada y se para al principio de la misma.
  - `quit`: Termina la ejecución del depurador

make: Herramienta para el mantenimiento de programas

- Facilita el proceso de generación y actualización de un programa.
- Determina automáticamente qué partes de un programa deben recompilarse ante una actualización de algunos módulos y las recompila.
- Para realizar este proceso, `make` debe conocer las dependencias entre los ficheros: Un fichero debe actualizarse si alguno de los que depende es más nuevo.
- `make` consulta un fichero (`Makefile`) que contiene las reglas que especifican las dependencias de cada fichero objetivo y los mandatos para actualizarlo.
- Formato de una regla:

  ```
  objetivo: dep1 dep2 ...
  TABmandato1
  TABmandato2
  TABmandato3
  ...
  ```

- Formato del mandato:

  ```
  make objetivo
  ```

- Que implica:
  1. Encontrar la regla correspondiente a objetivo

2. Tratar sus dependencias como objetivos y actualizarlas recursivamente.
3. Actualizar objetivo si alguna de las dependencias es más actual.

- Sin argumentos `make` activa la primera regla.
- Se pueden definir también macros:

  `NOMBRE=VALOR`

- Para acceder al valor de una macro:

  `$(NOMBRE) o ${NOMBRE}`

- Existen macros especiales: $@ corresponde con el nombre del objetivo.
- Se pueden especificar reglas basadas en la extensión de un fichero. Algunas de ellas están predefinidas (p.ej. la que genera el .o a partir del .c).

Ejemplo

```
CC=gcc
CFLAGS=-g
OBJS2=prac2.o aux2.o

all: prac1 prac2

prac1: prac1.o aux1.o
	gcc -g -o prac1 prac1.o aux1.o

prac2: $(OBJS2)
	${CC} ${CFLAGS} -o $@ ${OBJS2}

prac1.o prac2.o: prac.h

cleano:
	rm -f prac1.o aux1.o ${OBJS2}
```